УТВЕРЖДЕН

АДМГ.20134-01 95 01-ЛУ

ПРИКЛАДНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ «АВРОРА ЦЕНТР»

Руководство разработчика

Подсистема Сервис уведомлений

АДМГ.20134-01 95 01

Листов 50

2023

АННОТАЦИЯ

Настоящий документ является руководством разработчика подсистемы Сервис уведомлений (ПСУ), входящей в состав прикладного программного обеспечения «Аврора Центр» АДМГ.20134-01 релиз 4.0.0 (далее – ППО).

ППО может быть использовано, но не ограничиваться, в системах и объектах, описание которых приведено в документе «Руководство администратора» АДМГ.20134-01 91 01.

ПРИМЕЧАНИЯ:

1. Подробная информация о составе и назначении ППО, а также требования к условиям выполнения приведены в документе АДМГ.20134-01 91 01.

2. Подробная информация об особенностях резервного копирования приведена в документе «Рекомендации по резервному копированию» АДМГ.20134-01 91 02.

Настоящий документ содержит инструкции для разработчика мобильных приложений (МП), выполняющих отправку push-уведомлений.

СОДЕРЖАНИЕ

1. Общие принципы работы 4
1.1. Работа с тестовым Сервером приложений ПСУ 4
1.2. Получение доступа к тестовому серверу 6
1.3. Плагин для эмуляции push-уведомлений 7
2. Описание работы 9
2.1. Подготовка к работе 9
2.1.1. Общие указания 9
2.1.2. Настройка push-демона 10
2.2. Разработка МП 13
2.2.1. Общее описание 13
2.2.2. Работа с push-уведомлениями (для ОС версии ниже 4.0) 22
2.2.3. Работа с push-уведомлениями (для ОС от версии 4.0 и выше) 32
2.3. Отправка push-уведомлений 33
2.4. Запрос на Сервер приложений ПСУ 34
2.5. Пример кода сервера приложений 37
2.5.1. Код на языке Python 37
2.5.2. Код на языке Java 39
3. Ограничения 47
Перечень терминов и сокращений 48

1. ОБЩИЕ ПРИНЦИПЫ РАБОТЫ

ПРИМЕЧАНИЕ. Функциональность получения push-уведомлений доступна в операционной системе (ОС) Аврора, начиная с версии 3.2.2.

Если на момент отправки push-уведомления мобильное устройство (МУ) недоступно, МП получит push-уведомление после подключения к сети, обеспечивающей доступ к Серверу приложений ПСУ.

Для использования push-уведомлений в своих системах разработчикам необходимо расширить функциональность МП и сервера приложений, отправляющего push-уведомления.

ПРИМЕЧАНИЕ. Не рекомендуется с помощью push-уведомлений осуществлять передачу конфиденциальной и чувствительной информации в незащищенных сетях, т.к. протокол взаимодействия МУ с Сервером приложений ПСУ не подразумевает передачу конфиденциальной информации.

При разработке МП доступны следующие способы тестирования функциональности push-уведомлений:

– подключение к тестовому серверу;

– использование плагина для эмуляции Сервера приложений ПСУ.

1.1. Работа с тестовым Сервером приложений ПСУ

Для того, чтобы использовать в разработке тестовый Сервер приложений ПСУ для отправки push-уведомлений, необходимо выполнить следующие шаги:

1) Получить доступ к тестовому Серверу приложений ПСУ:

– направить запрос в техническую поддержку предприятия-разработчика dev-support@omp.ru;

– получить ключи для доступа к тестовому Серверу приложений ПСУ;

– использовать ключи при взаимодействии сервера приложения с тестовым Сервером приложений ПСУ.

2) Создать и установить МП на МУ:

– установить зависимости;

– передать в МП `applicationId`, полученный от поддержки разработчиков;

– зарегистрироваться в push-демоне с помощью `applicationId`, получив в ответ на запрос `registrationId` для тестового Сервера приложений ПСУ.

3) Организовать передачу push-уведомлений сервера приложения через тестовый Сервер приложений ПСУ:

– передать `registrationId` серверу приложения доверенным способом от МП. Это позволит серверу приложения отправлять таргетированные push-уведомления к своим МП на конкретном МУ;

– получить токен доступа на тестовом Сервере приложений ПСУ через сервер приложения;

– передать через сервер приложения push-уведомления, используя токен доступа и идентификатор связки приложение – устройство `registrationId`.

Процесс регистрации МП приведен на рисунке (Рисунок 1).

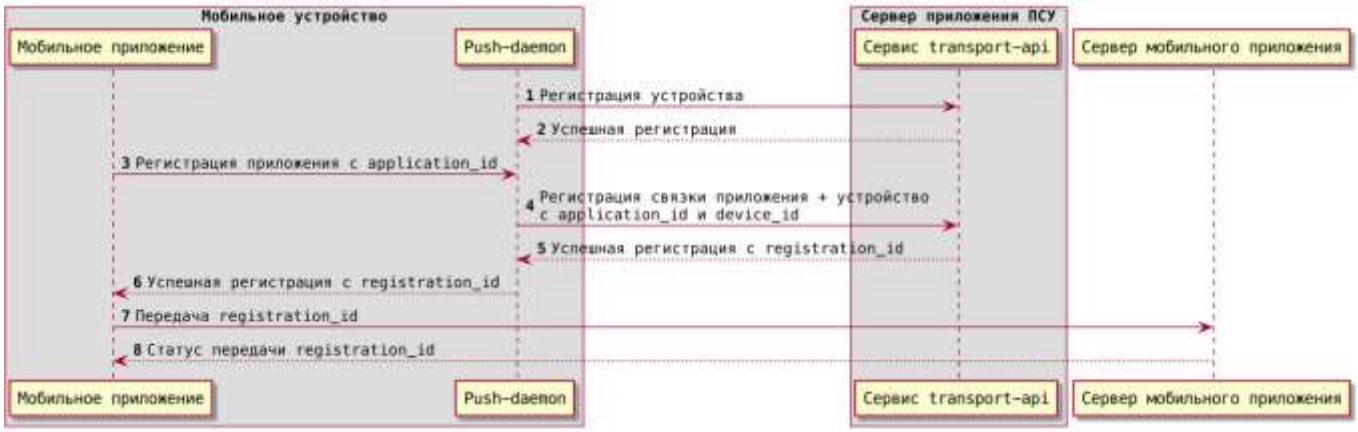

Рисунок 1

Запрос на регистрацию МП должен происходить при каждом его запуске, т.к `registrationId` имеет срок жизни, не зависящий от МП.

1.2. Получение доступа к тестовому серверу

ПРИМЕЧАНИЕ. Для тестирования сервера приложения, который отправляет push-уведомления, и МП, которое получает push-уведомления, необходимо использовать тестовый Сервер приложений ПСУ.

Для получения настроек подключения к тестовому Серверу приложений ПСУ необходимо направить письмо в техническую поддержку предприятия-разработчика: dev-support@omp.ru, указав название организации, которой требуется предоставить доступ. В ответном письме будут направлены адрес и порт, а также файл с настройками Сервера приложений ПСУ и настройками МП. Настройки Сервера приложений ПСУ выдаются в единственном экземпляре на организацию, при этом МП (каждое из которых имеет настройки), может быть несколько.

Настройки необходимо указать в конфигурации сервера приложений, который после прохождения аутентификации получает токен доступа и отправляет push-уведомления через Сервер приложений ПСУ.

Список настроек для подключения к тестовому Серверу приложений ПСУ:

– `project_id` – уникальный идентификатор проекта;

– `push_public_address` – адрес тестового Сервера приложений ПСУ, на который будут отправляться запросы;

– `api_url` – адрес API тестового Сервера приложений ПСУ;

– `client_id` – аккаунт учетной записи разработчика МП в подсистеме безопасности тестового Сервера приложений ПСУ, обычно совпадает с ID проекта;

– `scopes` – настройки области видимости (для OAuth 2);

– `audience` – настройки аудитории для токена доступа (для OAuth 2);

– `token_url` – адрес получения токена авторизации для запросов к тестовому Серверу приложений ПСУ;

– `key_id` – идентификатор приватного ключа RSA для передачи сообщений между сервером приложений и Сервером приложений ПСУ;

– `private_key` – приватный ключ RSA для передачи сообщений между сервером приложений и тестовым Сервером приложений ПСУ.

Для МП доступна настройка следующих параметров:

– `application_id` – уникальный идентификатор МП;

– `project_id` – уникальный идентификатор проекта.

ПРИМЕЧАНИЕ. `application_id` необходимо использовать для регистрации в push-демоне на МУ. Каждому `application_id` может соответствовать только 1 приложение.

1.3. Плагин для эмуляции push-уведомлений

В Аврора SDK предусмотрена возможность протестировать работу стороннего МП, отправляющего push-уведомления, на эмуляторе.

Для настройки плагина push-уведомлений необходимо выполнить следующие действия:

1) Сгенерировать сертификат `cert.pem` и ключ `key.pem`, выполнив команду `openssl` в терминале в системе разработчика. Вместо многоточий допускается заполнить поля в опции `-subj` собственными значениями или опустить. Ограничения на данные значения отсутствуют.

Пример:

```
openssl req -x509 -newkey rsa:4096 -keyout key.pem -out cert.pem -days
365 -nodes -subj
"/C=RU/ST=.../L=.../O=.../OU=.../CN=.../emailAddress=..."
```

2) В Аврора IDE запустить машину сборки Аврора Build Engine, нажав кнопку запуска, расположенную в нижнем левом углу окна Аврора IDE;

3) Создать файл настроек для службы push-уведомлений: «Параметры» -> «Push-уведомления» -> вкладка «Служба» -> «Файл настроек» -> «Создать»;

4) На той же вкладке «Служба» в разделе «Конфигурация» указать ключи из шага 1 в полях «Файл сертификата» и «Файл ключа»;

5) При необходимости добавить значение по умолчанию для полей «Данные» и «Заголовок» во вкладке «Значение по умолчанию для уведомлений»;

6) Нажать кнопку «Применить»;

7) Установить на эмуляторе пакет `push-example` из системных репозиториев. Для этого необходимо подключиться к эмулятору и выполнить команду `pkcon`:

```
ssh -p 2223 -i ~/AuroraOS/vmshare/ssh/private_keys/Aurora_OS-Emulator-latest/nemo nemo@localhost
pkcon install push-example
```

8) Открыть МП `push-example` на эмуляторе;

9) В Аврора IDE открыть нижнюю вкладку «Push-уведомления» или нажать сочетание клавиш Alt+9;

10) Выбрать идентификатор МУ;

11) Ввести идентификатор МП в поле справа от идентификатора МУ. Для `push-example` это `testApplication`;

12) Ввести текст уведомления и нажать кнопку «Отправить push-уведомление».

Плагин для эмуляции push-уведомлений позволяет отправлять уведомления не только на тестовое, но и на собственные МП. Для этого необходимо выполнить следующие действия:

– запустить МП;

– в Аврора IDE во вкладке «Push-уведомления» указать в качестве идентификатора `applicationId`, с которым МП регистрируется в push-демоне. Идентификатор для эмулятора может быть любым.

ПРИМЕЧАНИЕ. Подробное описание процесса разработки собственного МП для push-уведомлений приведено в настоящем документе.

2. ОПИСАНИЕ РАБОТЫ

ВНИМАНИЕ! Ошибки, которые могут выдаваться в ходе работы, приведены в подразделе 2.4.

2.1. Подготовка к работе

2.1.1. Общие указания

Для получения push-уведомлений на МУ используется push-демон, являющийся системным компонентом ОС Аврора.

ПРИМЕЧАНИЕ. При разработке МП следует убедиться, что в Аврора SDK и на МУ имеются необходимые пакеты.

Для входа в Аврора Build Engine необходимо выполнить команду:

```
ssh -p 2222 -i ~/AuroraOS/vmshare/ssh/private_keys/engine/mersdk
mersdk@localhost
```

Для установки пакетов (если они отсутствуют) в Аврора Build Engine необходимо выполнить команду:

```
zypper in push-daemon-libs push-daemon-devel
```

Для входа на МУ необходимо выполнить команду (`device ip` следует уточнить в настройках МУ; `username` для версий Аврора ОС, начиная с 4.0 – `defaultuser`, для версий ниже – `nemo`):

```
ssh <username>@<device ip>
```

Для МУ необходимо выполнить команду установки библиотек:

```
pkcon install push-daemon push-daemon-libs
```

Для проверки наличия пакета в Аврора Build Engine необходимо выполнить команду:

```
zypper search <имя пакета>
```

Пример команды для МУ:

```
pkcon search <имя пакета>
```

2.1.2. Настройка push-демона

2.1.2.1. Для ОС версии ниже 4.0

Для указания адреса сервера, который будет использоваться push-демоном для соединения, необходимо выполнить следующие действия:

– создать директорию `/etc/xdg/push-daemon`/, если она отсутствует;

– создать в данной директории конфигурационный файл `push-daemon.conf` или использовать уже имеющийся, если данный файл создан заранее;

– в конфигурационном файле в группе `[Remote]` указать IP-адрес или DNS-имя сервера в поле «ip», а также порт в поле «port»;

– после обновления конфигурации перезагрузить push-демон с помощью команды:

```
systemctl --user restart push-daemon
```

Пример конфигурационного файла `/etc/xdg/push-daemon/push-daemon.conf` с IP-адресом и портом:

```
[Remote]
ip=192.168.2.1
port=10001
```

IP-адрес, порт и DNS-имя сервера можно получить в поддержке предприятия-разработчика.

После перезапуска push-демона в системных журналах (директория `/var/log`) отобразятся строки следующего вида, означающие что соединение с сервером прошло успешно:

```
Sep 11 20:50:59 INOIP4903 push-daemon[23649]: [D] unknown:0 -
"Attempting to connect to PNS on 192.168.2.1:10001"
.............
Sep 11 20:50:59 INOIP4903 push-daemon[23649]: [D] unknown:0 - Got
connection responce for "{bd73dad4-e12c-41d9-a10f-5286eeea7fea}"
```

2.1.2.2. Для ОС от версии 4.0 и выше

Для указания адреса сервера, который будет использоваться push-демоном для соединения, необходимо в режиме администратора `devel-su` выполнить команду:

```
gdbus call -y -d \
ru.omprussia.PushDaemon -o /ru/omprussia/PushDaemon -m \
ru.omprussia.PushDaemon.SetNetworkConfiguration \
"{'address':<'DNS-имя или IP-адрес сервера'>, \
'port':<Порт сервера>}"
```

ПРИМЕЧАНИЕ. D-Bus-вызов заменяет выполнение следующих действий:

– редактирование файла `/var/lib/push-daemon/push-daemon.conf`;

– перезапуск push-демона вручную.

Полный список параметров конфигурации приведен в таблице (Таблица 1).

Таблица 1

Параметр	Тип	Описание
`address`	string	Адрес Сервера приложений ПСУ
`port`	uint	Порт Сервера приложений ПСУ
`crlUpdateInterval`	uint	Период обновления `crl` в секундах (не рекомендуется использовать малые значения, т.к. это сильно нагрузит систему. Оптимально – раз в сутки или реже)
`connectionValidation`	bool	Необходимость валидировать соединение с сервером. При отключении все `ssl` ошибки будут проигнорированы
`crlValidation`	bool	Необходимость проверять отозванность сертификата. При отключении все стандартные проверки (срок действия, валидность цепочки и т.д.) проводятся, при этом не проверяется, был ли сертификат отозван. Данная настройка имеет смысл только при включенном `connectionValidation`

Параметр	Тип	Описание
`hostNameValidation`	string	Уровень валидации имени хоста, указанного в сертификате: – `none` – имя не проверяется; – `relaxed` – допускает наличие * в имени (поддержка поддоменов, например, «*.example.ru»); – `strict` – имя, указанное в сертификате, должно полностью совпадать с адресом сервера. Данная настройка имеет смысл только при включенном `connectionValidation`

ПРИМЕЧАНИЕ. IP-адрес, порт и DNS-имя сервера можно получить в технической поддержке предприятия-разработчика.

После вызова метода конфигурации push-демона в системных журналах (директория `/var/log`) отобразятся строки следующего вида, означающие, что соединение с сервером прошло успешно:

```
Jun 29 14:22:07 INOIP4903 push-daemon[1987]: push.network: Device
online, trying to establish connection
Jun 29 14:22:07 INOIP4903 push-daemon[1987]: push.network: Now ready
to connect. Attempt  0
Jun 29 14:22:07 INOIP4903 push-daemon[1987]: push.network: Connecting
to server on "push.someserver.ru:10999"
Jun 29 14:22:07 INOIP4903 push-daemon[1987]: push.network: Creating
protocol connection message
Jun 29 14:22:07 INOIP4903 push-daemon[1987]: push.network: Sending
protocol connection message
Jun 29 14:22:07 INOIP4903 push-daemon[1987]: push.network: Message
sent. Currently stored 0 messages
Jun 29 14:22:07 INOIP4903 push-daemon[1987]: push.network: Handling
connection message responce
Jun 29 14:22:07 INOIP4903 push-daemon[1987]: push.network: Connection
established
Jun 29 14:22:07 INOIP4903 push-daemon[1987]: push.network: Creating
ping message
Jun 29 14:22:07 INOIP4903 push-daemon[1987]: push.network: Message
stored in queue. Currently stored 1 messages
Jun 29 14:22:07 INOIP4903 push-daemon[1987]: push.network: Sending
stored message
Jun 29 14:22:07 INOIP4903 push-daemon[1987]: push.network: Creating
activation message
Jun 29 14:22:07 INOIP4903 push-daemon[1987]: push.network: Activating:
"dbbe16bb-9249-405e-afd6-a9bb6892713a"
```

Данный метод не рекомендуется к использованию. При наличии необходимости использовать старый метод перезапуск push-демона должен осуществляться в режиме администратора `devel-su` командой:

```
systemctl restart push-daemon
Конфигурация, задаваемая через DBus-вызов, имеет приоритет над
конфигурацией, находящейся в /etc/xdg/push-daemon/push-daemon.conf.
Проверить конфигурацию можно следующей командой (или посмотрев файл
/var/lib/push-daemon/push-daemon.conf):
gdbus call -y -d \
ru.omprussia.PushDaemon -o /ru/omprussia/PushDaemon -m \
ru.omprussia.PushDaemon.GetNetworkConfiguration
```

2.2. Разработка МП

2.2.1. Общее описание

Для полноценной работы push-демона МП должно поддерживать работу в фоновом режиме, т.е. процесс должен выполняться и при закрытии графического интерфейса.

В ПСУ поддерживается работа с МП, в которых указанный режим реализован с помощью сервиса D-Bus.

ПРИМЕЧАНИЕ. Функционал получения push-уведомлений доступен в ОС Аврора, начиная с версии 3.2.2, при этом необходимо убедиться в наличии требуемых пакетов в SDK и на МУ при разработке.

2.2.1.1. Для ОС версии ниже 4.0

Основной класс, используемый для работы с push-уведомлениями, – это `Sailfish::PushNotifications::Client`.

В начале работы МП необходимо выполнить следующие действия:

– установить `applicationId` с помощью метода `Sailfish::PushNotifications::Client::setApplicationId();`

– вызвать метод `Sailfish::PushNotifications::Client::startHandler()`.

Значение `applicationId` необходимо получить в технической поддержке предприятия-разработчика. Данный идентификатор можно вынести в конфигурацию МП или программно запрашивать у ПСУ до начала процесса регистрации.

ПРИМЕЧАНИЕ. Каждому `applicationId` может соответствовать только одно МП.

На основе `applicationID` push-демон определяет соответствующий ему `deviceID`. После этого при вызове метода `registrate()` эта пара отправляется на сервер, где производится проверка, существует ли такая пара. Если нет, то генерируется `registrationId`, который возвращается МП через push-демон. Если пара уже была зарегистрирована, то возвращается существующий `registrationId`, поэтому `registrationId` является постоянным для связки МП и МУ.

В случае неактивности пользователя в течение длительного времени будет отправлен сигнал `clientInactive`, при этом если МП работает в фоновом режиме и не выполняется никаких активных процессов, то оно должно завершить работу в целях экономии заряда аккумулятора. В дальнейшем для возобновления работы МП потребуется запустить его заново.

Каждое push-уведомление содержит поля для заголовка, тело сообщения и объект с данными, который представляет собой набор пар «ключ-значение». Структура с описанием push- уведомления:

```
struct Push
{
    QString data; // строка, содержащая сериализованный JSON-объект
    QString title; // заголовок сообщения
    QString message; // тело сообщения
    QString action; // зарезервировано для будущего использования
    QString image; // зарезервировано для будущего использования
    QString sound; // зарезервировано для будущего использования
    quint32 categoryId = 0; // зарезервировано для будущего
использования
};
```

ПРИМЕЧАНИЕ. МП доступны 3 атрибута:

– `title`;

– `message`;

– `data`.

Следует обратить внимание, что data является сериализованным в строку JSON-объектом. Формат JSON-объекта в data может быть произвольным. Атрибут data, в частности, можно использовать для указания типа push-уведомления — text, command и т.д., например «{«type»: «command», «text»: «value»}».

Список этих структур Sailfish::PushNotifications::PushList возвращается с сигналом Sailfish::PushNotifications::Client::notifications (см. API Sailfish::PushNotifications::Client). Каждое push-уведомление, полученное таким образом, в дальнейшем может быть выведено как уведомление (п. 2.2.2).

API клиента Sailfish::PushNotifications::Client для работы с push-демоном состоит из методов и сигналов.

Основные методы:

– bool startHandler() – обработчик push-уведомлений (должен вызваться при каждом запуске МП). Возвращает true, если обработчик был успешно запущен;

– void setApplicationId(const QString &applicationId) - устанавливает объекту класса Client applicationId МП, должен быть вызван непосредственно перед вызовом startHandler;

Пример:

```
 bool Application::start(const QStringList &arguments)
{
    auto applicationId = getApplicationId();
    m_client->setApplicationId(applicationId); // m_client - это
объект класса `Sailfish::PushNotifications::Client`
    if (!m_client->startHandler()) {
        qWarning() << "Failed to start push client, can not start
handler";
        return false;
    }
    if (arguments.indexOf(QStringLiteral("--gui")) != -1) {
        startGui();
    }
    return true;
}
```

– QString applicationId() – возвращает установленный в объекте класса Client applicationId;

Пример:

```
qDebug() << "Current client applicationId:" << m_client-
>applicationId(); // m_client - это объект класса
Sailfish::PushNotifications::Client
```

– `void registrate()` – отправляет запрос на Сервер приложений ПСУ на регистрацию МП (следует вызывать при каждом запуске МП);

– `bool getConnectionStatus()` - возвращает статус соединения. true, если МУ и Сервер приложений ПСУ соединены, false – в противоположном случае.

Основные сигналы:

– `void connectionStatusChanged(bool status)` – передается при изменении состояния соединения push-демона и Сервера приложений ПСУ. `Status` имеет значение true, если push-демон и Сервер приложений ПСУ соединены, false – в противоположном случае;

Пример:

```
connect(m_client,
&Sailfish::PushNotifications::Client::connectionStatusChanged, this,
[](bool status){
    qDebug() << "Daemon is now connected to PNS: " << status;
});
```

– `void registrationId(const QString &registrationId)` – передается при получении объектом класса `Client registrationId`, содержащий идентификатор МП. Данный идентификатор должен быть отправлен на Сервер приложений ПСУ для реализации возможности отправки push-уведомлений;

Пример:

```
connect(m_client,
&Sailfish::PushNotifications::Client::registrationId, this,
[this](const QString &registrationId){
    qDebug() << QString("Registration is successful for %1.
RegistrationId: %2").arg(applicationId()).arg(registrationId);
    setRegistrationId(registrationId);
});
```

– `void registrationError()` – передается в случае неудачной регистрации;

Пример:

```
connect(m_client,
&Sailfish::PushNotifications::Client::registrationError, this,
[this]() {
    qWarning() << "Registration error for " << applicationId();
});
```

– `void notifications(const Aurora::PushNotifications::PushList &pushList)` - передается при получении объектом класса `Client` push-уведомлений. `pushList` содержит список сообщений;

Пример:

```
connect(m_client, &Sailfish::PushNotifications::Client::notifications,
this, [this](const Sailfish::PushNotifications::PushList &pushList){
    for (const Sailfish::PushNotifications::Push &push : pushList) {
        qInfo() << "Push title" << push.title;
        qInfo() << "Push message" << push.message;
    }
});
```

– `void clientInactive()` - передается в случае, если МП не взаимодействовало с push-уведомлениями длительное время. Если МП запущено в фоновом режиме, его рекомендуется выключить;

Пример:

```
connect(m_client,
&Sailfish::PushNotifications::Client::clientInactive, this, [this](){
    if (!m_guiStarted) {
        qInfo() << "Handling inactivity in background mode";
        QGuiApplication::instance()->quit();
        return;
    }
    qInfo() << "Ignoring inactivity signal";
});
```

2.2.1.2. Для ОС от версии 4.0 и выше

ПРИМЕЧАНИЕ. Функционал перенесен из `namespace Sailfish::PushNotifications` в `Aurora::PushNotifications`.

Основной класс, используемый для работы с push-уведомлениями, – это `Aurora::PushNotifications::Client`.

В начале работы МП необходимо установить `applicationId` с помощью метода `Aurora::PushNotifications::Client::setApplicationId();`

Значение `applicationId` необходимо получить в технической поддержке предприятия-разработчика. Данный идентификатор можно вынести в конфигурацию МП или программно запрашивать у ПСУ до начала процесса регистрации.

ПРИМЕЧАНИЕ. Каждому `applicationId` может соответствовать только одно МП.

На основе `applicationID` push-демон выполняет поиск соответствующего ему `registrationId`, после чего при вызове метода `registrate()` эта пара отправляется на сервер, где производится проверка ее актуальности. Если пара не актуальна (или если `registrationId` пустой в случае первой регистрации), то генерируется новый `registrationId`, который возвращается МП через push-демон. Если же пара уже была актуальна, то возвращается существующий `registrationId`, поэтому `registrationId` является постоянным для связки МП и МУ.

В случае неактивности пользователя в течение длительного времени будет отправлен сигнал `clientInactive`, при этом если МП работает в фоновом режиме и не выполняется никаких активных процессов, то оно должно завершить работу в целях экономии заряда аккумулятора. В дальнейшем для возобновления работы МП его потребуется запустить заново.

Каждое push-уведомление содержит поля для заголовка, тело сообщения и объект с данными, который представляет собой набор пар «ключ-значение». Структура с описанием push-уведомления:

```
struct Push
{
    QString data; // строка, содержащая сериализованный JSON-объект
    QString title; // заголовок сообщения
    QString message; // тело сообщения
    QString action; // зарезервировано для будущего использования
    QString image; // зарезервировано для будущего использования
    QString sound; // зарезервировано для будущего использования
    quint32 categoryId = 0; // зарезервировано для будущего
использования
};
```

ПРИМЕЧАНИЕ. МП доступны 3 атрибута:

– `title`;

– `message`;

– `data`.

`Data` является сериализованным в строку JSON-объектом. Формат JSON-объекта в `data` может быть произвольным. Атрибут `data`, в частности, можно использовать для указания типа push-уведомления — `text`, `command` и т. д., например «{«type»: «command», «text»: «value»}».

Список этих структур `Aurora::PushNotifications::PushList` возвращается с сигналом `Aurora::PushNotifications::Client::notifications` (см. API `Aurora::PushNotifications::Client`). Каждое push-уведомление, полученное таким образом, в дальнейшем может быть выведено как уведомление (п. 2.2.2).

API клиента `Aurora::PushNotifications::Client` для работы с push-демоном состоит из методов и сигналов.

Основные методы:

– `void setApplicationId(const QString &applicationId)` - устанавливает объекту класса `Client applicationId` МП, должен быть вызван непосредственно перед вызовом `startHandler`.

Пример:

```
 bool Application::start(const QStringList &arguments)
{
    auto applicationId = getApplicationId();
    m_client->setApplicationId(applicationId); // m_client - это
объект класса `Aurora::PushNotifications::Client`
    if (!m_client->startHandler()) {
        qWarning() << "Failed to start push client, can not start
handler";
        return false;
    }
    if (arguments.indexOf(QStringLiteral("--gui")) != -1) {
        startGui();
    }
    return true;
}
```

– `QString applicationId()` - возвращает установленный в объекте класса `Client applicationId`.

Пример:

```
qDebug() << "Current client applicationId:" << m_client-
>applicationId(); // m_client - это объект класса
Aurora::PushNotifications::Client
```

– `void registrate()` – отправляет запрос на Сервер приложений ПСУ на регистрацию МП (следует вызывать при каждом запуске МП);

– `bool isPushSystemReady()` – возвращает статус готовности push-демона:

- `true`, если push-демон может обрабатывать запросы МП;
- `false` – в противоположном случае;

– `int error()` – позволяет получить код последней ошибки регистрации;

– `QString errorMessage()` – позволяет получить текстовое описание последней ошибки регистрации.

Основные сигналы:

– `void pushSystemReadinessChanged(bool status)` – передается при изменении состояния push-демона. `Status` имеет значение true, если МУ способно получать push-уведомления, false – в противоположном случае;

Пример:

```
connect(m_client,
&Aurora::PushNotifications::Client::pushSystemReadinessChanged, this,
[](bool status){
    qDebug() << "Daemon can receive push notifications: " << status;
});
```

– `void registrationId(const QString &registrationId)` – передается при получении объектом класса `Client` `registrationId`, содержащего идентификатор МП. Данный идентификатор должен быть отправлен на Сервер приложений ПСУ для реализации возможности отправки push-уведомлений;

Пример:

```
connect(m_client, &Aurora::PushNotifications::Client::registrationId,
this, [this](const QString &registrationId){
    qDebug() << QString("Registration is successful for %1.
RegistrationId: %2").arg(applicationId()).arg(registrationId);
    setRegistrationId(registrationId);
});
```

– `void registrationError()` – передается в случае неудачной регистрации;

Пример:

```
connect(m_client,
&Aurora::PushNotifications::Client::registrationError, this, [this]()
{
    qWarning() << "Registration error for " << applicationId();
});
```

– `void notifications(const Aurora::PushNotifications::PushList &pushList)` – передается при получении объектом класса `Client` push-уведомлений. `PushList` содержит список сообщений;

Пример:

```
connect(m_client, &Aurora::PushNotifications::Client::notifications,
this, [this](const Aurora::PushNotifications::PushList &pushList){
    for (const Aurora::PushNotifications::Push &push : pushList) {
        qInfo() << "Push title" << push.title;
        qInfo() << "Push message" << push.message;
    }
});
```

– `void clientInactive()` – передается в случае, если МП не взаимодействовало с push-уведомлениями длительное время. Если МП запущено в фоновом режиме, его рекомендуется выключить;

Пример:

```
connect(m_client, &Aurora::PushNotifications::Client::clientInactive,
this, [this](){
    if (!m_guiStarted) {
        qInfo() << "Handling inactivity in background mode";
        QGuiApplication::instance()->quit();
        return;
    }
    qInfo() << "Ignoring inactivity signal";
});
```

2.2.2. Работа с push-уведомлениями (для ОС версии ниже 4.0)

Пример стороннего МП `omp-reference-push-example` демонстрирует использование API `Push daemon`, получение и обработку push-уведомлений, а также их отображение. Пример полностью опубликован на ресурсе gitlab.com/AuroraOS.

Архитектура МП включает следующие элементы:

– `main.cpp` — стартовая точка МП;

– `Application` — класс, реализующий клиентское API для работы с push-уведомлениями. Унаследован от `QObject`;

– `ControlService` — класс, реализующий сервис для контроля работы МП в фоновом режиме/на переднем плане с помощью D-Bus-сервиса и аргументов командной строки. Унаследован от `QDBusAbstractAdaptor`;

– вспомогательные классы и qml-код для графического интерфейса (в настоящем документе не приводятся, могут быть определены произвольно).

2.2.2.1. Объявление МП

Фрагмент кода класса, отвечающего за взаимодействие API push-демона и интерфейса МП:

```
#pragma once

#include <QObject>

#include <push_client.h>
#include <nemonotifications-qt5/notification.h>

class Application : public QObject
{
    Q_OBJECT
    Q_PROPERTY(QString applicationId READ applicationId CONSTANT)
    Q_PROPERTY(QString registrationId READ registrationId WRITE
setRegistrationId NOTIFY registrationIdChanged)

public:
    explicit Application(QObject *parent = nullptr);

    bool start(const QStringList &arguments);

    QString applicationId() const;

public slots:
    void startGui();

    QString registrationId() const;
    void setRegistrationId(const QString& registrationId);

    void processRegistrationError();
    void processRegistrationId(const QString &registrationId);
    void processNotifications(const
Aurora::PushNotifications::PushList &pushList);

signals:
    void registrationIdChanged(const QString& registrationId);

private slots:
    void handlePushClientInactivity();

private:
    bool m_guiStarted = false;
    QString m_registrationId;

    Aurora::PushNotifications::Client *m_client;

    QString getApplicationId() const;
};
```

Подключение библиотек `push_client.h` и `nemonotifications-qt5/notification.h` позволяет использовать API `Aurora::PushNotifications`.

В конструкторе происходит подключение сигналов `Aurora::PushNotifications::Client` к слотам `Application`:

```
Application::Application(QObject *parent)
    : QObject(parent),
      m_client(new Aurora::PushNotifications::Client(this))
{
    connect(m_client,
&Aurora::PushNotifications::Client::registrationId, this,
&Application::processRegistrationId);
    connect(m_client,
&Aurora::PushNotifications::Client::registrationError, this,
&Application::processRegistrationError);
    connect(m_client,
&Aurora::PushNotifications::Client::notifications, this,
&Application::processNotifications);
    connect(m_client,
&Aurora::PushNotifications::Client::clientInactive, this,
&Application::handlePushClientInactivity);
    connect(m_client,
&Aurora::PushNotifications::Client::connectionStatusChanged, [](bool
status)
    {
        qDebug() << "New daemon network status" << status;
    });
}
```

2.2.2.2. Идентификатор МП

Для успешной работы МП необходимо создать файл `/usr/share/omp-reference-push-example/applicationid` и добавить в него идентификатор МП, который был получен вместе с ключами доступа к Серверу приложений ПСУ. Для этого требуется выполнить на МУ команду:

```
echo "идентификатор" > /usr/share/omp-reference-push-
example/applicationid
```

Следует убедиться, что у файла достаточно прав, чтобы пользователь мог читать этот файл. Для выдачи прав файлу можно выполнить на МУ команду:

```
chmod 644 /usr/share/omp-reference-push-example/applicationid
```

В классе `Application` идентификатор из `/usr/share/omp-reference-push-example/applicationid` можно считать в методе `getApplicationId()` следующим образом:

```
QString Application::getApplicationId() const
{
    QFile applicationIdFile("/usr/share/omp-reference-push-
example/applicationid");

    qDebug() << "Try to get clinent id from file " <<
applicationIdFile.fileName() << " with permissions " <<
applicationIdFile.permissions();

    if (applicationIdFile.exists() &&
applicationIdFile.open(QIODevice::ReadOnly))
    {
        auto data = applicationIdFile.readAll().trimmed();
        qDebug() << "Read from file and use " << data << " as
application id";
        return data;
    }
    else
    {
        qDebug() << "Did not find a file " <<
applicationIdFile.fileName() <<  " or can no open it or have no
permissions. Use omp_reference_push_example as application id";
        return QStringLiteral("omp_reference_push_example");
    }
}
```

Альтернативным вариантом указания `applicationId` является задать его непосредственно в коде МП или передать от ПСУ.

В методе `start()` класса `Application` считывается идентификатор МП, который затем передается push-демону:

```
bool Application::start(const QStringList &arguments)
{
    auto applicationId = getApplicationId();
    m_client->setApplicationId(applicationId); // Регистрация в Push-
daemon'е
    if (!m_client->startHandler())
    {
        qWarning() << "Failed to start push client, can not start
handler";
        return false;
    }
    if (arguments.indexOf(QStringLiteral("--gui")) != -1)
    {
```

```
        startGui();
    }
    return true;
}
```

Также при старте обрабатываются аргументы командной строки `arguments`. Аргумент `–gui` может служить сигналом запуска графического интерфейса в методе `startGui()`:

```
void Application::startGui()
{
    if (m_guiStarted)
    {
        return;
    }
    qDebug() << "Daemon network status" << m_client-
>getConnectionStatus();
    m_client->registrate();
    qInfo() << "Starting GUI";
    m_guiStarted = true;
    auto view = AuroraApp::createView();
    view->rootContext()->setContextProperty("ApplicationInstance",
this);
    // Здесь можно указать другие настройки графического интерфейса
    view->setSource(AuroraApp::pathTo(QStringLiteral("qml/omp-
reference-push-example.qml"))); // Здесь можно указать путь к главной
qml-странице
    view->show();
}
```

2.2.2.3. Регистрационный идентификатор

Результат регистрации в push-демоне будет обработан в слоте `processRegistrationId()` и методе `setRegistrationId()` у `Application`:

```
void Application::processRegistrationId(const QString &registrationId)
{
    qDebug() << QString("Registration is successful for %1.
RegistrationId: %2")
                .arg(applicationId())
                .arg(registrationId);

    setRegistrationId(registrationId);
}

void Application::setRegistrationId(const QString &registrationId)
{
    if (registrationId != m_registrationId)
    {
        m_registrationId = registrationId;
```

```
        emit registrationIdChanged(registrationId);
    }
}
```

После запуска МП пытается зарегистрироваться на Сервере приложений ПСУ и в случае успеха получает идентификатор регистрации. В консоли `/journalctl` отобразится сообщение следующего вида:

```
[D] Application::processRegistrationId:98 - "Registration is
successful for omp_reference_p_btut7giq44tbluarkai1. RegistrationId:
62bba07e-1000-4d6e-ad8b-47935e1b7d64"
```

`registrationId` необходимо отправить ПСУ, чтобы он мог отправлять push-уведомления стороннему МП через сервер push-уведомлений.

2.2.2.4. Push-уведомления

Получив push-уведомление, МП сначала проверяет, не находится ли МП на переднем плане, а затем выполняет анализ поля данных push-уведомления и поиск ключа `mtype` в полученном JSON.

Если `mtype` равно:

– `action` — уведомление не будет отображаться;

– `notify` — будет отображен только верхний баннер;

– `system_notify` — уведомление будет отображаться только в центре уведомлений.

При любом другом значении, а также при пустом значении или отсутствии `mtype` будут отображаться 2 push-уведомления (баннер и центр уведомлений).

Выбор способа отображения push-уведомления реализован в методе `processNotifications` класса `Application`:

```
void Application::processNotifications(const
Aurora::PushNotifications::PushList &pushList)
{
    qDebug() << QStringLiteral("Application %1: got %2
pushes").arg(applicationId()).arg(pushList.count());

    for (const Aurora::PushNotifications::Push &push : pushList)
    {
        qDebug() << "{";
        qDebug() << "Title: " << push.title;
        qDebug() << "Data: " << push.data;
```

```
        qDebug() << "Message: " << push.message;
        qDebug() << "}";
        // Здесь можно отобразить сообщение в графическом интерфейсе,
например, добавить как элемент модели списка.
    }

    auto state = qGuiApp->applicationState(); // qGuiApp - глобальный
объект Qt, указывающий на данное приложение.

    if (state != Qt::ApplicationActive) // Если приложение работает в
фоновом режиме, Push-уведомления будут отображены как уведомления на
экране.
    {
        for (const Aurora::PushNotifications::Push &push : pushList)
        {
            Notification notification;
            notification.setCategory(QString("common"));

            auto jsonDcoument =
QJsonDocument::fromJson(push.data.toUtf8());
            auto type =
jsonDcoument.object().value("mtype").toString();

            if (type == QStringLiteral("action"))
            {
                continue;
            }

            if (type == QStringLiteral("notify"))
            {
                notification.setPreviewSummary(push.title);
                notification.setPreviewBody(push.message);
            }
            else if (type == QStringLiteral("system_notify"))
            {
                notification.setSummary(push.title);
                notification.setBody(push.message);
            }
            else
            {
                notification.setSummary(push.title);
                notification.setBody(push.message);
                notification.setPreviewSummary(push.title);
                notification.setPreviewBody(push.message);
            }
            notification.publish();
        }
    }
}
```

2.2.2.5. Сервис для управления МП

Для управления МП, способным как работать в фоновом режиме, так и запускать графический интерфейс, создается класс `ControlService`, который наследуется от `QDBusAbstractAdaptor`:

```
class ControlService : public QDBusAbstractAdaptor
{
    Q_OBJECT
    Q_CLASSINFO("D-Bus Interface", DBUS_INTERFACE)

public:
    explicit ControlService(QObject *parent);

    static bool isApplicationRunning();
    static int updateApplicationArgs(const QStringList &arguments);

    static QString const service;
    static QString const path;
    static QString const interface;

    bool start();

signals:
    void startGui();

public slots:
    void handleArguments(const QStringList &arguments);
};
```

Название сервиса, путь и интерфейс можно задать как константы:

```
QString const ControlService::service = QStringLiteral(DBUS_SERVICE);
QString const ControlService::path = QStringLiteral(DBUS_PATH);
QString const ControlService::interface =
QStringLiteral(DBUS_INTERFACE);
```

В методе `start()` регистрируется D-Bus-сервис:

```
bool ControlService::start()
{
    auto session = QDBusConnection::sessionBus();

    if (!session.registerObject(path, parent()))
    {
        qWarning() << "Cannot register ControlService D-Bus path: " <<
path << ". Error: "<< session.lastError().message();
        return false;
    }

    qDebug() << "Registered ControlService D-Bus path: " << path;
```

```
    if (!session.registerService(service))
    {
        qWarning() << "Cannot register ControlService D-Bus service: "
<< service << ". Error: " << session.lastError().message();
        return false;
    }

    qDebug() << "Registered ControlService D-Bus service:" << service;

    return true;
}
```

Метод `updateApplicationArgs(const QStringList &arguments)` использует этот сервис, чтобы применить для МП аргументы командной строки, вызвав `handleArguments(const QStringList &arguments)`:

```
int ControlService::updateApplicationArgs(const QStringList
&arguments)
{
    auto message = QDBusMessage::createMethodCall(service, path,
interface, QStringLiteral("handleArguments"));
    message.setArguments(QList<QVariant>() << arguments);
    QDBusReply<void> reply =
QDBusConnection::sessionBus().call(message);

    if (!reply.isValid())
    {
        qWarning() << reply.error().message();
    }

    return 0;
}
```

Метод `handleArguments(const QStringList &arguments)` отправляет сигнал о необходимости открыть графический интерфейс, если в аргументах командной строки имеется `--gui`:

```
void ControlService::handleArguments(const QStringList &arguments)
{
    if (arguments.indexOf(QStringLiteral("--gui")) != -1)
    {
        qDebug() << "Ask for run gui";

        emit startGui();
    }
}
```

Сервис создается и запускается в `main.cpp`:

```
int main(int argc, char *argv[])
```

```
{
    auto application = AuroraApp::application(argc, argv);

    QStringList applicationArguments = application->arguments();
    applicationArguments.removeFirst();

    if (ControlService::isApplicationRunning())
    {
        qDebug() << "Application already running, use this runned
application";
        return
ControlService::updateApplicationArgs(applicationArguments);
    }
    else
    {
        qDebug() << "Starting application " << application-
>arguments();

        auto controlService = new ControlService(application);

        if (!controlService->start())
        {
            return 0;
        }

        auto pushExample = new Application(application);
        QObject::connect(controlService, &ControlService::startGui,
pushExample, &Application::startGui);

        pushExample->start(applicationArguments);

        return application->exec();
    }
}
```

Сначала проверяется, не был ли сервис уже запущен. В этом случае у него только обновляются аргументы командной строки. Иначе создается и запускается новый сервис, а также МП, которое получает аргументы командной строки.

2.2.2.6. Дополнительные настройки конфигурации

Для МП аргументы командной строки можно задать, например, в desktop-файле:

```
Exec=omp-reference-push-example --gui %u
```

В pro-файле необходимо указать, что МП использует Qt D-Bus и дополнительные библиотеки: `pushclient` и `nemonotifications-qt5`:

```
QT += dbus
```

```
PKGCONFIG += \
    pushclient \
    nemonotifications-qt5

INCLUDEPATH += /usr/include/pushclient
```

Кроме того, в этом же файле можно задать константы для имени, пути и интерфейса собственного D-Bus-сервиса:

```
DEFINES += DBUS_SERVICE=\\\"ru.omprussia.reference_push_example\\\"
DEFINES += DBUS_PATH=\\\"/ru/omprussia/reference_push_example\\\"
DEFINES += DBUS_INTERFACE=\\\"ru.omprussia.reference_push_example\\\"
```

ПРИМЕЧАНИЕ. После регистрации МП в push-демоне оно отправляет на сервер ПСУ уникальный `registrationId`, при этом на сервере приложений необходим процесс, способный получать и сохранять `registrationId` для дальнейшей таргетированной отправки push-уведомлений.

2.2.3. Работа с push-уведомлениями (для ОС от версии 4.0 и выше)

Пример, приведенный выше, остается актуальным за исключением следующих ключевых отличий:

– изменение API (п. 2.2.2);

– для доступа к push-демону МП необходимо разрешение `PushNotifications`, которое можно задать в поле `Permissions` секции `X-Sailjail` desktop-файла:

```
[X-Sailjail]
Permissions=PushNotifications
```

– ограничение на имена D-Bus шин, которые может поднимать МП. Доступны только имена вида `orgname.appname`, где `orgname` и `appname` задаются в секции `X-Sailjail` desktop-файла:

```
[X-Sailjail]
OrganizationName=com.org
ApplicationName=app
```

– механизм включения GUI был инвертирован. Если раньше требовалось включать GUI при наличии ключа, заданного разработчиком, то теперь при наличии в аргументах ключа `/no-gui`, передаваемого push-демону, указанная необходимость отсутствует;

– для того, чтобы МП могло быть запущено в push-демоне с таким ключом в `Exec`, поле desktop-файла должно содержать аргумент `%u`:

```
Exec=omp-reference-push-example %u
```

2.3. Отправка push-уведомлений

Push-уведомления отправляются через API Сервера приложений ПСУ, который защищен протоколом OAUTH2 OpenID Connect согласно JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication. При этом используется метод аутентификации `private_key_jwt`.

При регистрации проекта на Сервере приложений ПСУ разработчику будут направлены конфигурационные настройки МП и сервера приложений. В конфигурации представлены параметры, необходимые для настройки клиента протокола авторизации OpenID Connect и формирования URL для отправки push-уведомлений, приведенные в подразделе 1.1.

Этапы отправки push-уведомлений:

– получение токена, позволяющего в течение времени его жизни отправлять push-уведомления;

– непосредственно отправка push-уведомления.

Для получения токена необходимо выполнить неавторизованный запрос по адресу из параметра `token_url` (подраздел 1.1).

В параметрах запроса необходимо указать следующие данные:

– тип авторизации `"client_credentials"`;

– `audience` из файла настроек сервера;

– `scope` из файла настроек сервера;

– client_assertion_type как строку "urn:ietf:params:oauth:client-assertion-type:jwt-bearer";

– client_assertion как сформированный самостоятельно токен jwt.

Из ответа следует считать атрибуты access_token и expires_in, которые возвращают токен и время его действия.

Примеры кода на Python и Java для получения токена доступа приведены в подразделе 2.5.

Примеры кода приведены на портале разработчиков ОС Аврора: https://community.omprussia.ru/documentation/software_development/guides/push/server.html#example.

2.4. Запрос на Сервер приложений ПСУ

Для отправки push-уведомления необходимо сформировать запрос на Сервер приложений ПСУ, указанный в api_url (подраздел 1.1).

К адресу Сервера приложений ПСУ необходимо добавить фрагмент с указанием project_id:

```
POST /api/v1/projects/{project_id}/messages
```

Заголовок Content-Type должен иметь значение application/json.

Обязательные параметры тела запроса приведены в таблице (Таблица 2).

Таблица 2

Параметр	Тип	Описание
target	String	Указывается registrationId, полученный из МП после его регистрации на Сервере приложений ПСУ
type	Enum (String)	Поддерживается значение device
ttl	String	Время жизни push-уведомления, например, 5h30m. Может быть задано только в следующих единицах: s — секунды, h — часы, m — минуты
notification	OBJ Notification	Содержание push-уведомления

Характеристики объекта `Notification` приведены в таблице (Таблица 3).

Таблица 3

Параметр	Тип	Описание
`title`	`string`	Заголовок (до 512 символов). Обязательное поле
`message`	`string`	Тело сообщения (до 2048 символов). Обязательное поле
`data`	`Object`	Объект для параметров ключ-значение, можно передавать любые необходимые МП поля (до 1024 байт). Опциональное поле

Возможные коды ошибок приведены в таблице (Таблица 4).

Таблица 4

Код и статус ошибки	Описание	Действия для устранения ошибки
400 Bad Request	ttl limit is exceeded	Необходимо проверить допустимое время жизни push-уведомления в соответствии с настоящим документом
	unsupported message type	Необходимо проверить доступные типы доставки уведомлений в соответствии с настоящим документом
	invalid notification title length	Необходимо проверить доступные ограничения на создание уведомлений в соответствии с настоящим документом
	invalid notification message length	Необходимо проверить доступные ограничения на создание уведомлений в соответствии с настоящим документом
	invalid target	Необходимо проверить доступные значения в соответствии с настоящим документом
	target not found (указанный registration_id не найден)	`registration_id` необходимо получить после регистрации МУ и МП на Сервере приложений ПСУ

Код и статус ошибки	Описание	Действия для устранения ошибки
401	Client authentication failed, the provided client JSON Web key is expired (не удалось выполнить аутентификацию. Срок действия предоставленного ключа истек)	Необходимо обратиться к администратору Сервиса уведомлений с просьбой выпустить новый ключ безопасности проекта и прислать его, после чего следует установить новый ключ и повторить запрос
401 Unauthorized	Истек срок действия токена авторизации	Необходимо получить новый токен авторизации
	Client authentication failed, the provided client JSON Web key is expired (не удалось выполнить аутентификацию. Срок действия предоставленного ключа истек)	Необходимо обратиться к администратору Сервиса уведомлений с просьбой выпустить новый ключ безопасности проекта и прислать его, после чего следует установить новый ключ и повторить запрос
403 Forbidden	Отсутствуют необходимые права доступа для выполнения запроса	Необходимо обратиться к системному администратору для назначения прав
413 Request Entity Too Large	Общий размер JSON-объекта в теле сообщения превышает 4 кб	Необходимо уменьшить размер JSON-объекта в теле сообщения
429 Too Many Requests	Достигнут лимит количества запросов в секунду	Необходимо сократить количество запросов в секунду для проекта
500 Internal Server Error	Внутренняя системная ошибка Сервера приложений ПСУ	Необходимо обратиться к системному администратору
504 Gateway Timeout	Инфраструктура Сервиса уведомлений недоступна	Необходимо обратиться к системному администратору

Пример запроса:

```
POST https://global-push.domain/api/projects/project-from-ui-
bthsc2iq44tso869omt0/messages
Content-Type: application/json
{
    "target": "442125d4-6041-4f72-ab4b-1e5fd3ad389a",
    "ttl": "2h",
    "type": "device",
    "notification": {
        "title": "some title",
        "message": "some message",
        "data": {
            "action": "command",
            "another_key": "value"
        }
    }
}
```

Пример ответа в случае успеха имеет следующий вид:

```
{
    "expiredAt": "2020-10-06T07:44:43.967576Z",
    "id": "1526c09c-1ca4-48ea-8357-e1b8aa2f3fc3",
    "notification": {
        "data": {
            "action": "command",
            "another_key": "value"
        },
        "message": "some message",
        "title": "some title"
    },
    "status": "",
    "string": "", "target": "442125d4-6041-4f72-ab4b-1e5fd3ad389a",
    "type": "device"
}
```

2.5. Пример кода сервера приложений

2.5.1. Код на языке Python

Тестовый пример на языке программирования Python демонстрирует все описанные ранее шаги. Пример полностью доступен на ресурсе gitlab.com/AuroraOS.

Программа считывает настройки проекта из текстового файла с именем `config`, получает `token_endpoint` от API push-сервиса `well-known`, делает запрос на получение токена доступа и в заключительной части отправляет push-уведомление (необходимо подставить реальный `registrationId`, полученный с МУ).

До запуска примера необходимо рядом с файлом кода создать конфигурационный файл с именем `config`, который содержит настройки, полученные при регистрации на Сервере приложений ПСУ. Настройки в файле должны быть указаны в определенном формате, в частности:

– каждая настройка расположена на отдельной строке;

– каждая настройка имеет вид (ключ: значение).

Пример конфигурационного файла:

```
Адрес Push-сервера:https://aurora-partner-push-
test.ompcloud.ru/push/public
scope:openid offline message:update
audience:ocs-auth-public-api-gw,ocs-push-public-api-gw
clientId:<id>
privateKeyId:<id>
ID проекта:<id>
privateKey:-----BEGIN RSA PRIVATE KEY-----
<key>
-----END RSA PRIVATE KEY-----
```

Полный пример кода сервера приложений доступен на ресурсе gitlab.com/AuroraOS.

Запустить пример можно при помощи следующей команды:

```
python3 ./main.py
```

Программа выводит результат в следующем формате:

```
Project options:
    push_public_address: "url"
    project_id: "id"
    client_id: "id"
    audience: "ocs-auth-public-api-gw ocs-push-public-api-gw"
    scope: "openid offline message:update"
    private_key_id: "id"
    private_key: "b'-----BEGIN RSA PRIVATE KEY-----
    \n<key>==\n-----END RSA PRIVATE KEY-----\n'"
Got token_endpoint:
    "http://ocs-push-dev.ompcloud.ru/auth/public/oauth2/token"
Result:
    response: {'access_token': '<token>', 'expires_in': 3599, 'scope':
'openid offline message:update', 'token_type': 'bearer', 'expires_at':
1601966682}
    access_token: <token>
Send push message result:
```

```
    response: "{'expiredAt': '2020-10-06T07:44:43.967576Z', 'id':
'1526c09c-1ca4-48ea-8357-e1b8aa2f3fc3', 'notification': {'data':
{'action': 'command', 'another_key': 'value'}, 'message': 'somem
message', 'title': 'some title'}, 'status': '', 'string': '',
'target': '7cefc0d1-b262-4bcc-a959-497c7bfd38f4', 'type': 'device'}"
    id: "1526c09c-1ca4-48ea-8357-e1b8aa2f3fc3"
```

2.5.2. Код на языке Java

Тестовый пример на Java демонстрирует ту же функциональность и в полном виде доступен на ресурсе gitlab.com/AuroraOS.

Программа считывает настройки проекта из yaml-файла с именем, заданным в аргументах командной строки, получает `token_endpoint` от API push-сервиса `well-known`, делает запрос на получение токена доступа и в заключительной части отправляет push-уведомление (необходимо подставить реальный `registrationId`, полученный с МУ).

До запуска примера необходимо создать конфигурационный файл с именем `config`, который содержит настройки, полученные при регистрации на Сервере приложений ПСУ. Настройки в файле должны быть указаны в определенном формате:

– настройки Сервера приложений ПСУ должны быть в разделе `server`, клиентского МП — в разделе `client`;

– каждая настройка расположена на отдельной строке;

– каждая настройка имеет вид (ключ: значение).

Пример конфигурационного файла:

```
server:
  projectId: <id>
  pushAddress: https://aurora-partner-push-
test.ompcloud.ru/push/public
  authAddress: https://aurora-partner-push-
test.ompcloud.ru/push/public
  clientId: <id>
  scope: openid offline message:update
  audience: ocs-auth-public-api-gw ocs-push-public-api-gw
  privateKey: |
    -----BEGIN RSA PRIVATE KEY-----
    <key>
    -----END RSA PRIVATE KEY-----

  privateKeyId: <key>
```

```
client:
  applicationId: <id>
```

Назначение классов в примере сервера приложений:

– классы `Server`, `Client` и `Config` — чтение и хранение настроек клиента МП и Сервера приложений ПСУ;

– класс `TokenFetcher` — запрос и получение токена от Сервера приложений ПСУ;

– класс `TestPushClient` — стартовая точка МП, загрузка настроек из файла, получение токена с помощью `TokenFetcher`, проверка токена и отправка push-уведомления.

Фрагменты кода `server` и `client`:

```
public class Server {
    private String projectId;
    private String clientId;
    private String privateKeyId;
    private String privateKey;
    private String scope;
    private String audience;
    private String pushAddress;
    private String authAddress;
    ...
    <методы для установки и получения значений полей и метод toString>
}

public class Client {
    private String applicationId;
    ...
    <методы для установки и получения значений поля и метод toString>
}
```

Класс `Config` использует Java-библиотеку `SnakeYAML` для чтения конфигурационного yaml-файла:

```
import org.yaml.snakeyaml.Yaml;
import org.yaml.snakeyaml.constructor.Constructor;
import org.yaml.snakeyaml.introspector.PropertyUtils;
import org.yaml.snakeyaml.representer.Representer;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.InputStream;

public class Config {

    private Server server;
    private Client client;
```

```
    public static Config load(String configPath) throws
FileNotFoundException {
        InputStream inputStream = new FileInputStream(configPath);
        Constructor constructor = new Constructor(Config.class);
        PropertyUtils propUtils = new PropertyUtils();
        propUtils.setAllowReadOnlyProperties(true);
        Representer repr = new Representer();
        repr.setPropertyUtils(propUtils);

        Yaml yaml = new Yaml(constructor, repr);
        return yaml.load(inputStream);
    }
    ...
    <методы для установки и получения значений полей и метод toString>
}
```

Класс `TokenFetcher` использует библиотеки:

– `Unirest` для работы с сетью;

– `Jose4j` для генерации JSON Web Token (JWT), который необходим для безопасного обмена сообщениями с Сервером приложений ПСУ;

– `Security` для работы с RSA-ключом.

```
import kong.unirest.*;
import kong.unirest.json.JSONObject;
import org.jose4j.jws.AlgorithmIdentifiers;
import org.jose4j.jws.JsonWebSignature;
import org.jose4j.jwt.JwtClaims;
import org.jose4j.jwt.NumericDate;
import org.jose4j.jwt.consumer.JwtConsumer;
import org.jose4j.jwt.consumer.JwtConsumerBuilder;
import org.jose4j.lang.JoseException;
import ru.omp.push.example.config.Config;
import java.security.KeyFactory;
import java.security.PrivateKey;
import java.security.spec.PKCS8EncodedKeySpec;
import java.util.Base64;
import java.util.Date;

public class TokenFetcher {
    private final String pushServer;
    private final String clientId;
    private final String scope;
    private final String privateKey;
    private final String privateKeyId;
    private final String audience;
    public TokenFetcher(Config cfg) {
        this.audience = cfg.getServer().getAudience();
        this.clientId = cfg.getServer().getClientId();
```

```
        this.privateKey = cfg.getServer().getPrivateKey();
        this.privateKeyId = cfg.getServer().getPrivateKeyId();
        this.pushServer = cfg.getServer().getPushAddress();
        this.scope = cfg.getServer().getScope();
    }
    public boolean validateToken(String accessToken) throws Exception
{
        JwtConsumer jwtConsumer = new JwtConsumerBuilder()
                .setSkipSignatureVerification()
                .setExpectedAudience(this.audience.split(" "))
                .build();
        JwtClaims jwtClaims =
jwtConsumer.processToClaims(accessToken);
        return
!jwtClaims.getExpirationTime().isBefore(NumericDate.fromMilliseconds((
new Date()).getTime() + 60000));
    }
    public String authenticate() throws Exception {
        HttpResponse<JsonNode> wellKnownResponse =
Unirest.get(pushServer + "/api/v1/.well-known").asJson();
        String wellKnownResponseJs =
wellKnownResponse.getBody().getObject().get("token_endpoint").toString
();
        PrivateKey signingKey = takeSigningKey();
        String privateKeyJWT = generatePrivateKeyJWT(signingKey,
wellKnownResponseJs);
        return getAccessToken(privateKeyJWT, wellKnownResponseJs);
    }
    protected String getAccessToken(String privateKeyJWT, String
wellKnownResponseJs) throws Exception {
        HttpResponse<JsonNode> jsonResponse =
Unirest.post(wellKnownResponseJs)
                .field("grant_type", "client_credentials")
                .field("client_assertion_type",
"urn:ietf:params:oauth:client-assertion-type:jwt-
bearer").field("client_assertion", privateKeyJWT)
                .field("scope", scope)
                .field("audience", audience)
                .asJson();
        if (jsonResponse.getStatus() != 200) {
            throw new Exception("Authentication failed");
        }
        JSONObject accessToken = jsonResponse.getBody().getObject();
        return accessToken.get("access_token").toString();
    }
    protected String generatePrivateKeyJWT(PrivateKey rsaPrivate,
String audience) throws Exception {
        JwtClaims claims = new JwtClaims();
        claims.setIssuer(clientId);
        claims.setSubject(clientId);
        claims.setAudience(audience);
        claims.setExpirationTimeMinutesInTheFuture(5);
```

```
        claims.setIssuedAtToNow();
        claims.setGeneratedJwtId();
        return getJwt(privateKeyId, rsaPrivate, claims);
    }
    protected String getJwt(String keyId, PrivateKey rsaPrivate,
JwtClaims claims) throws JoseException {
        JsonWebSignature jws = new JsonWebSignature();
        jws.setPayload(claims.toJson());
        jws.setKey(rsaPrivate);
        jws.setKeyIdHeaderValue(keyId);

jws.setAlgorithmHeaderValue(AlgorithmIdentifiers.RSA_USING_SHA256);
        return jws.getCompactSerialization();
    }
    protected PrivateKey takeSigningKey() throws Exception {
        String clearPrivateKey = privateKey
                .replace("-----BEGIN RSA PRIVATE KEY-----", "")
                .replaceAll("\n", "")
                .replace("-----END RSA PRIVATE KEY-----", "");
        byte[] decoded = Base64.getDecoder().decode(clearPrivateKey);
        return KeyFactory.getInstance("RSA").generatePrivate(new
PKCS8EncodedKeySpec(decoded));
    }
}
```

TestPushClient также использует Unirest для работы с сетью. Путь к конфигурационному файлу и registrationId МП берутся из первого и второго аргумента командной строки соответственно.

```
import kong.unirest.ContentType;
import kong.unirest.HttpResponse;
import kong.unirest.JsonNode;
import kong.unirest.Unirest;
import ru.omp.push.example.auth.TokenFetcher;
import ru.omp.push.example.config.Config;

import java.io.FileNotFoundException;

public class TestPushClient {
    private final Config cfg;
    private final TokenFetcher tokenFetcher;
    private String getToken() throws Exception {
        return this.tokenFetcher.authenticate();
    }
    private int makeRequest(String accessToken, String registrationId)
{
        String url = cfg.getServer().getPushAddress()
                + "/api/v1/projects/" + cfg.getServer().getProjectId()
+ "/messages";
        String json = "{\n" +
                "  \"target\": " + registrationId + ",\n" +
                "  \"ttl\": \"2h\",\n" +
```

```
                "  \"type\": \"device\",\n" +
                "  \"notification\": {\n" +
                "    \"title\": \"some title\",\n" +
                "    \"message\": \"some message\",\n" +
                "    \"data\": {\n" +
                "      \"action\": \"command\",\n" +
                "      \"another_key\": \"value\"\n" +
                "    }\n" +
                "  }\n" +
                "}";
        HttpResponse<JsonNode> jsonResponse =
                Unirest.post(url)
                        .header("Content-Type",
ContentType.APPLICATION_JSON.getMimeType())
                        .header("Authorization", "Bearer " +
accessToken)
                        .body(new JsonNode(json)).asJson();
        int code = jsonResponse.getStatus();
        String response = jsonResponse.getBody().toPrettyString();
        System.out.println("HTTP Code: " + code);
        System.out.println("Response: " + response + "\n");
        return code;
    }
    public TestPushClient(String configPath) throws
FileNotFoundException {
        Config cfg = Config.load(configPath);
        System.out.println(cfg);
        this.cfg = cfg;
        this.tokenFetcher = new TokenFetcher(cfg);
        // for test purposes in case of self-signed server
certificates
        Unirest.config().verifySsl(false);
    }
    public static void main(String[] args) {
        try {
            if (args.length == 0) {
                System.out.println("Use:\n\t java -jar
./build/libs/sample-push-client-java-1.0.0.jar <path to YAML config
file> <registrationId>");
                System.exit(1);
            }
            TestPushClient client = new TestPushClient(args[0]);
            String accessToken = client.getToken();
            System.out.println("ACCESS TOKEN: " + accessToken);
            if (args.length > 1) {
                String registrationId = args[1];
                if (registrationId.trim().length() > 0) {
                    // Validate token for expiration.
                    // This code block is offline validation of token
before real request to send push
                    System.out.println("\nValidating token...");
                    if
(client.tokenFetcher.validateToken(accessToken)) {
                        System.out.println("Token VALID");
```

```
                    } else {
                        System.out.println("TOKEN not valid. Please
ask for new");
                        accessToken = client.getToken();
                    }
                    System.out.println("\nSending push-message:");
                    int code = client.makeRequest(accessToken,
registrationId);
                    if (code == 401) {
                        System.out.println("Need to refresh token and
make request ");
                        /*
                          In real life in case of 401 we have to get
new access token and make request again,
                          but it depends on application server
strategy, i.e.:
                             accessToken = client.getToken();
                             code = client.makeRequest(accessToken,
registrationId);
                         */
                    } else if (code < 200 || code > 299) {
                        System.out.println("FAILED");
                    } else {
                        System.out.println("SUCCESS");
                    }
                }
            }
        } catch(Exception e) {
            System.out.println(e.getMessage());
            e.printStackTrace();
        }
    }
}
```

Для запуска примера из каталога `sample-push-client-java` текущего проекта необходимо выполнить команду:

```
java -jar ./build/libs/sample-push-client-java-1.0.0.jar <path to
config.yml> <registrationId>
```

Программа выведет ответ в следующем формате:

```
ru.omprussia.omp.push.ru.omp.push.example.config.Config{server=Server{
projectId='<id>', clientId='<id>', privateKeyId='<id>',
privateKey='<key>', scope='openid offline message:update',
audience='ocs-auth-public-api-gw ocs-push-public-api-gw',
pushAddress='http://ocs-push-dev.ompcloud:8009/push/public',
authAddress='http://ocs-push-dev.ompcloud:8009/push/public'},
client=Client{applicationId='<id>'}}
ACCESS TOKEN:
<token>


Validating token...
```

```
Token VALID

Sending push-message:
HTTP Code: 202
Response: {
  "createdAt": "0001-01-01T00:00:00Z",
  "expiredAt": "2020-11-20T17:59:24.803836Z",
  "id": "56b7522f-cded-4f0a-8d63-bf17809e4b5f",
  "notification": {
    "createdAt": "0001-01-01T00:00:00Z",
    "data": {
      "action": "command",
      "another_key": "value"
    },
    "message": "some message",
    "title": "some title",
    "updatedAt": "0001-01-01T00:00:00Z"
  },
  "target": "cbba8bc0-4756-446c-8cce-5140bee8e49d",
  "type": "device",
  "updatedAt": "0001-01-01T00:00:00Z"
}

SUCCESS
```

Необходимо обратить внимание на следующее:

– перед тем, как делать запрос на отправку push-уведомлений, требуется проверить токен на актуальность (см. метод `validateToken` класса `TokenFetcher`);

– если запрос на отправку push-уведомления вернул HTTP code 401, токен невалидный. Требуется обновить его через метод `authenticate` класса `TokenFetcher` и произвести попытку отправить push-уведомление (такое поведение может зависеть от стратегии сервера приложений).

3. ОГРАНИЧЕНИЯ

В версии ОС Аврора 4-го семейства был расширен протокол взаимодействия Сервера приложений ПСУ и push-демона.

Общая логика взаимодействия осталась прежней, однако изменение протокола произошло ввиду изменения взаимодействия push-демона с компонентами управления приложениями в ОС Аврора.

Для версии 1 push-демона было необходимо запускать GUI, а в версии 2 push-демона запуск GUI не требуется ввиду наличия используемого ключа `nogui`.

Соответственно, для обработки push-уведомлений без запуска GUI необходимо реализовать поддержку соответствующего режима в самом приложении.

ПРИМЕЧАНИЕ. Дополнительные ограничения приведены в документе АДМГ.20134-01 91 01.

ПЕРЕЧЕНЬ ТЕРМИНОВ И СОКРАЩЕНИЙ

Используемые в настоящем документе термины и сокращения приведены в таблице (Таблица 5).

Таблица 5

Термин/ Сокращение	Расшифровка
Аврора SDK	Среда разработки Аврора, включающая в себя следующие компоненты: – Аврора IDE; – Аврора Emulator; – Аврора Build Engine
МП	Мобильное приложение
МУ	Мобильное устройство
ОС	Операционная система
Предприятие-разработчик	Общество с ограниченной ответственностью «Открытая мобильная платформа» (ООО «Открытая мобильная платформа»)
ППО	Прикладное программное обеспечение «Аврора Центр»
ПСУ	Подсистема Сервис уведомлений
API	Application Programming Interface — программный интерфейс приложения, описание способов (набор классов, процедур, функций, структур или констант), которыми одна компьютерная программа может взаимодействовать с другой программой
D-Bus	Desktop Bus — система межпроцессного взаимодействия, которая позволяет приложениям в операционной системе сообщаться друг с другом
DNS	Domain Name System — компьютерная распределенная система для получения информации о доменах
GUI	Graphical User Interface — разновидность пользовательского интерфейса, в котором элементы интерфейса (меню, кнопки, значки, списки), представленные пользователю на дисплее, исполнены в виде графических изображений

Термин/ Сокращение	Расшифровка
HTTP	HyperText Transfer Protocol — протокол прикладного уровня передачи данных (изначально в виде гипертекстовых документов). Основой HTTP является технология «клиент-сервер», т.е. предполагается существование потребителей (клиентов), которые инициируют соединение и посылают запрос, и поставщиков (серверов), ожидают соединения для получения запроса, производят необходимые действия и возвращают обратно сообщение с результатом
IP	Internet Protocol — основной протокол сетевого уровня, использующийся в сети Интернет и обеспечивающий единую схему логической адресации устройств в сети и маршрутизацию данных
JSON	Текстовый формат обмена данными, основанный на JavaScript
JWT	JSON Web Token
Push-уведомления	Текстовые сообщения, предназначенные для оперативной (мгновенной) доставки на МУ пользователей
RSA	Криптографический алгоритм с открытым ключом, основывающийся на вычислительной сложности задачи факторизации больших целых чисел
URL	Uniform Resource Locator — единообразный локатор (определитель местонахождения) ресурса

ЛИСТ РЕГИСТРАЦИИ ИЗМЕНЕНИЙ

Изм.	Номера листов (страниц)				Всего листов (страниц) в докум.	№ документа	Входящий № сопроводительного докум. и дата	Подп.	Дата
	изменённых	заменённых	новых	аннулированных					