


☐

I'm not robot


reCAPTCHA

Continue

Review How to receive push notifications using MZTT's Android mobile app Von I wrote earlier about MOTT as a technology to perform push notifications on a mobile phone. When I wrote this, I gave the example of an Android project. However, this was the first time I ever did android development, and although it was a good sample of Java MZTT, it was a bad Android sample - I knew nothing about how Android works as a platform. Since then I've written other Android MZTT apps such as the hackday app to click updates from websites on your phone and learned a lot about how to do it right. Ok... if not properly, at least a little better. But Google still directs people to my old, and probably useless, sample. So it's time that I share something more useful. I've put the full source to the sample below. (Note that I use the Java J2EE client library with ibm.com). We hope that the comments in it are quite clear, but here are some of the key points. Services vs. Activities Let's start with a simple one. The most obvious screw-up in my first attempt was not understanding the difference between activity and services. When you write the Android graphical interface, you expand the action. Only one action works on the phone at a time - an app that the user uses. When you switch to another app or close the app, the action is crushed. If you rotate your phone while you're running the app, the action is crushed and a new one is launched to implement the GUI in a new orientation. The point is that the activity is short-lived. Make them beautiful. Don't force them to do any lengthy weightlifting. Not having inposed any things from MOTT. Because he's going to be killed soon, without much warning. It doesn't matter if you kick off the background streams as they will be killed, too. My first sample of MOTT (which did everything in the activity) was completely unreliable. I created an MHT Client facility that creates a long-term TCP/IP connection. And then, if I turned the phone around a bit, my object MGTClunt would get the trash collected. Instead, I now use the service. Services are designed to carry out longer operations , multiple services can run at the same time. It doesn't matter if you GUI-up class gets trashed and trash collected - the service can continue to run together in the background. The MGT Client can survive in a service facility, and its constant TCP/IP connection can be open. Make this sticky service designed for the long term, but they don't stay forever. If the phone gets a low level of memory - usually because the user uses the app in the foreground (so as a high priority), which requires a lot of memory. It happens often - it's not an unusual thing that can happen once a blue moon. My experience is that every day with your Service. Several times. My service may bumble together for hours happily, but then it will be killed without warning. Returning a constant START_STICKY when your service is up and running, you tell Android that if it has to kill the service to free up valuable resources, then you want it to restart the service when the resource becomes available again. Persistent connection Here's what you get for free, but I thought it was interesting enough to note. When creating an MTTTClient object, one of the things it does is set up a long-term TCP/IP connection. When you receive a message, the callback method is called. When the user doesn't use their phone and it's turned off, Android starts turning things off to save battery life. My concern is that disabling the phone could break my connection, or stop the code waiting for incoming messages. Although I haven't found a good example of where this is documented, it seems that even if your phone sleeps, if the data is obtained on the connection, your code will be woken up and publishArrived called. The MTT customer library code that blocks the outlet will be revived when any data is available to the outlet. Keeping the connection alive - in a bad way Well... You almost get it for free. When you connect to a MGT server, one of the parameters is to keepAlive period - an agreement between the client and the server about how often the server should expect to hear from the client. If the server does not hear from the client during this period keepAlive, then it assumes that the client has left and closes the connection. The MGT Client knocks out the background stream, which is responsible for sending ping messages to the server often enough to keep the connection alive. I mentioned above that when a user doesn't use their phone and it's turned off, Android starts turning things off to save battery life. This includes a processor. And if the processor stops, that background stop. No background flow occurs, after the keepAlive period expires the server closes the connection. If you use RSMB, you see it in the server log. 20110130 234952.683 CWNAN0024I 1200 second time-out for customer 1296420495459774d56d6, ending the connection The easiest approach is to use Wake Lock. This is a way for the app to prevent the phone from shutting down. By taking action PARTIAL_WAKE_LOCK, you keep the processor running even if the user presses the power button. If you take PARTIAL_WAKE_LOCK before you make a MHT connection and save it, then the processor won't stop while your app is running, the standard background stream of MHTC Client never stops, and a long connection to the network is maintained. But... Erm... It also means that you have stopped the user from being able to turn off their phone! It's a little bit Since this has a big impact on the battery life of the phone. So I prefer to avoid it. Testing what happens when you turn off your phone - gotcha Some of my early attempts at the MOTT service for Android didn't take after the lock, but still worked fine. What I don't understand is that I had several apps installed on my phone that keep partial wakefulness locks all the time. Even though my app didn't stop the phone's CPU from turning, other apps were. And I benefited from it and didn't realize I had a problem. The ADB shell team dumpsys spitting out a ton of material on the connected phone, including a list of current wakefulness locks. (Search for 'mLocks.size' in the output). You have to do this several times to get a picture of what's going on - some apps will, quite legitimately, take a partial wake-up lock from time to time until they do something critical that they don't want to be interrupted. The app problems that take wake locks when they start, and keep it forever - are the ones that are always in the mLocks table in the output dumpsys. After uninstalling these apps, the results from my test were very different, and it was that showed me that I needed to do something to keep the link alive. Keeping the connection alive - in a good way the best approach might be to write a library of MHTT customers to work in a more Android-oriented way. But to be honest, I'm too lazy for this - so I use the existing Java J2EE client library, but in addition to that I'm serving my own keepAlive approach, assuming that the client library won't be good enough. AlarmManager provides a way to ask Android to wake up a device that sleeps at a specified time, run a specific method in your code. I schedule something to wake the phone before the current keepAlive period expires, long enough to send MHTT ping, after which it is free to go back to sleep. It doesn't wake up the phone screen if it's off - it just starts the processor, so my keep the live code can work. Note that the MZTT specification says that the server should give the client a grace period of half to keep the interval alive again (for example, if you keep alive for 20 minutes, then the server is required to cut off the client to 30 minutes after he last heard from him). This means I don't have to worry about getting my ping in just before the keepAlive interval - there's enough freedom in the specs that I can just schedule a ping for keepAlive seconds after the last interaction with the server. Do not rely on the connectionLost Library MTTTClient has a callback connectionLost, which must be called to inform the code if the connection If you have a long interval, then the client can wait for a long time before believing that not hearing from the server means that the connection is broken. It's This. callback, but in itself does not respond enough, especially for something like a phone where the state of the connection changes so often (e.g., moving from mobile data coverage and loss of communication, moving between Wi-Fi and mobile cellular data, etc.) Android provides a notification system to inform your app of these changes. In response to these notifications, I assume that my connection is no longer reliable and reconnect. If you use RSMB, you see it in the server log. 20110129 230652.613 CWNAN0033I Попытка подключения к слушателю 1883 получил от клиента 12963424184129774d56d6 по адресу 9 3.97.32.135:61055 20110129 230703.834 CWNAN0033I Попытка слушателя 1883 получено от клиента 12963424184129774d56d6 по адресу 82.132.248.199:58529 20110129 230703.834 CWNAN0034I Дубликат попытки подключения, полученный для идентификатора клиента 1296342418412977d56d6, окончание старейшего соединения Уважение запросов пользователя Android предоставляет возможность для пользователя, чтобы отключить использование фоновых данных приложениями. But it is not respected. The user may not untick background data to check to indicate that applications should not use the network in the background, and provides an API for applications to check this value. But apps can ignore this. This is not a good thing - users should be able to decide how their mobile is used. I check this value before I do anything, and set up the listener in case the value changes during the service. Create a unique customer ID, the MSTT Specification requires a unique customer ID for each customer. I use a unique device ID from my phone to create this. Annoyingly, it's not available on emulators, so with that I use timestamp - the current time in milliseconds from January 1, 1970. Note that the specification also determines the maximum allowable length of the customer ID, so I'll truncate this as needed. Assuming you'll spend time without communication it's more of a generic approach thing rather than a specific bit of code. But the service must assume that it will often be disabled, and must process reconnection whenever possible without requiring user intervention. I've seen apps (I'm looking at you, facebook!), which, if they're unable to connect to their server, just get stuck in an endless loop of retries - constantly trying and failing. On a personal note, I have more mobile phones than SIM cards. And I notice that with these apps installed, the battery life of the phone is worse when there is no SIM card in it than when there is, partly because these applications are constantly trying to communicate with the server without a network connection. Instead, discover when the network is not available and All. Don't spool, don't ping, don't keep trying or checking, just request notification when the connection is back, stop all keepalive keepalive run and sleep. And be prepared to get everything back up and running fast when the connection is available again. There is also a review of how you send server messages to the party. When you post a message, you can make the message saved. This means that if the client is unavailable, the server will store messages for him safely, and transfer them when the customer connects. This may not be suitable for all scenarios, but it is certainly something that I use a lot, and seems to fit well into mobile phones that are often disabled, the customer should not miss the messages published while he is in the black place. By reminding the user that you work in desktop window environments, users are used to knowing what works in their system - because they can see the windows they have open. On a mobile OS where only one application is open at the same time, it's harder to have a sense of what's running in the background. This is normal for services that stay running to do a specific thing, and will stop when they are completed. But for services like this that intend to run forever until manually stopped, the user should really remind what he is. In my example, I use a current notification in the state bar that will stay there whenever the service is running. Thus, the user is reminded that he uses some of the resources of his phone. You might think it might be annoying, so maybe it should be done optional - giving the user a way to hide the notification if they're happy to leave it running without a reminder. But even then, I would notice by default. It is likely that for most of the time the Service will not have an Activity user interface showing the latest data. Activity's user interface may be launched later and will want to have access to messages that were received during its launch. This means that the Service should probably store the messages it receives somewhere. This will depend on the application. Apps that handle very small amounts of data, such as updates and notifications that you don't need to save if the app and/or phone are restarted, etc., may be acceptable to store that data in a variable in the Service. That's what I do in the sample below - store it in a local hash-style. Apps that handle large amounts of data and/or need data to be stored, even if the app and/or phone are restarted, need to store data safely. There are a number of storage options available - choose the one that best suits the type/frequency/size, etc. of your data. Choosing the value of keep-alive, I keepAlive above - how often a client has to contact a server to keep the connection alive. It's a difficult decision. Desktop Desktop often, and you waste time battery life. If you ping too rarely and you may not notice if you lose touch before the next failed ping attempt. This is a trade-off between how time sensitive the data that your application processes is against the acceptable impact on battery life. But are there also perhaps a network of specific problems? How long will a mobile operator leave a idling connection open before they trash it? For example, it may be that there is no point in being kept alive for 40 minutes if the user's network operator will litter the connection after 20 minutes. I need to experiment with this more, but at the same time I use keep alive 20 minutes, which seems to be fairly reliable. Transfer config values is a small point, but it's worth thinking about how to provide the Service with the connection settings it needs, such as the server host's name. There are many ways to do this. Because I don't want users to enter these settings every time the service is up and running, I also need to save those settings. The easiest way to keep small amounts of config information so that it is available to the Service and modified by the Activity user interface is the way the example below does. Relaying received messages to interested user interfaces I outlined previously the main separation between back-end, long-term network connectivity implemented in the service, and the user interface implemented in the activity. There are different ways that they can communicate with each other. For this sample I use broadcasts. It's essentially a pub/sub local engine to the phone. Whenever the Service receives a message from the MZTT from the network, it broadcasts it locally over the phone. If Activity's user interface works, it will subscribe to these messages by creating BroadcastReceiver. When Action is not active, it may unregister these listeners. This means that the Service doesn't need to keep track of whether the Activity 9 user interface is working or not by removing the connection between the back and front ends of the app. My sample service - code with everything that waffling aside - is the code. My blog doesn't display the code very well, so you'd probably better copy and replicate it in your favorite IDE with Java coloring/formatting included. I put it there in the hope that it's useful. If you use it, I would appreciate thanks and/or link. © package dalelane.android.mqtt; import java.lang.ref.WeakReference; import java.util.Calendar; Import java.util.Date imports java.util.Enumeration; import java.util.Hashtable; Import android.app.AlarmManager; Import android.app.Notification; Import import android.app.PendingIntent; import android.app.Service; Imports Imports импортировать android.content.Context; импортировать android.content.Intent; импортировать android.content.IntentFilter; импортировать android.content.SharedPreferences; импортировать android.graphics.Color; импортировать android.net.ConnectivityManager; импортировать android.os.Binder; импортировать android.os.IBinder; импортировать android.os.PowerManager; импортировать android.os.PowerManager.WakeLock; импортировать android.provider.Settings; импортировать android.provider.Settings.Secure; импортировать android.util.Log; импорт com.ibm.mqtt.IMqttClient; импорт com.ibm.mqtt.MqttClient; импорт com.ibm.mqtt.MqttException; импорт com.ibm.mqtt.MqttNotConnectedException; импорт com.ibm.mqtt.MqttPersistence; импортировать com.ibm.mqtt.MqttPersistenceException; импорт com.ibm.mqtt.MqttSimpleCallback; Пример того, как реализовать клиент МЗТТ в Android, способный получать push-уведомления с сервера брокера сообщений МЗТТ. * * Dale Lane (dale.lane@gmail.com) * 28 Jan 2011 */ public class MQTTService extends Service implements MqttSimpleCallback { /******// * CONSTANTS */ /******// something unique to identify your app - used for stuff like accessing // application preferences public static final String APP_ID = com.dalelane.mqtt; // constants used to notify the Activity UI of received messages public static final String MQTT_MSG_RECEIVED_INTENT = com.dalelane.mqtt.MSGRECV; public static final String MQTT_MSG_RECEIVED_TOPIC = com.dalelane.mqtt.MSGRECVD_TOPIC; public static final String MQTT_MSG_RECEIVED_MSG = com.dalelane.mqtt.MSGRECVD_MSGBODY; // constants used to tell the Activity UI the connection status public static final String MQTT_STATUS_INTENT = com.dalelane.mqtt.STATUS; public static final String MQTT_STATUS_MSG = com.dalelane.mqtt.STATUS_MSG; // constant used internally to schedule the next ping event public static final String MQTT_PING_ACTION = com.dalelane.mqtt.PING; // constants used by status bar notifications public static final int MQTT_NOTIFICATION_ONGOING = 1; public static final int MQTT_NOTIFICATION_UPDATE = 2; // constants used to define MQTT connection status public enum MQTTConnectionStatus { INITIAL // первоначальный статус CONNECTING // попытка подключить CONNECTED // подключенный NOTCONNECTED_WAITINGFORINTERNET, // не может подключиться, потому что телефон // не имеет доступа в Интернет NOTCONNECTED_USERDISCONNECT, // Пользователь явно запросил // отключение NOTCONNECTED_DATADISABLED, // не может подключиться, потому что пользователь // отключил доступ к данным NOTCONNECTED_UNKNOWNREASON // не смог подключиться по какой-то причине // МТТ константы публичного статического int MAX_MQTT_CLIENTID_LENGTH // VARIABLES, используемые для поддержания штат () // MZTT Client Connections Private Connection mGTCONNECTIONStatus - MSTCONNECTIONStatus.INITIAL; // VARIABLES used to customize the MCHTT connection // // The name of the server that we receive push notifications from private string brokerHostName // taken from preferences // the name of the server to

