SYSTEM FOR MONITORING A FLEET OF ANDROID DEVICES USING TWO WAY MESSAGE QUEUING

Valeriu Manuel IONESCU
University of Pitesti, ROMANIA
valeriu.ionescu@upit.ro

Keywords: Android, monitoring, Message Queuing

Abstract: *This paper describes the design and implementation of a system that is able to monitor a fleet of vehicles. Rabbit MQ is used to send data between devices and the server. Tests are made to determine the problems of such implementation and recommendations are made to solve similar implementations.*

1. INTRODUCTION

Internet of things (IoT) is one of the most used keywords in technology at this moment. This summarizes the large number of devices that are connected to the internet in order to inform a server about the events that have occur, such as position, changes in pressure, temperature or simply information about a device's status. The trend is to make all devices "smart" and connected to the internet[1].

Monitoring large number of devices (a fleet), such as vehicles, is often necessary because of people want to know where their location, history and how to organize them better in the future. Keeping track of such a large number of devices can prove to be a difficult task, because the server or servers that receive the data from the devices must meet one or more of the following criteria (this is not a restrictive list): must have sufficient bandwidth; must have redundancy; are power efficient; are cost efficient; are available 24/7. One of the solution that easily meets the outlined criteria is to use servers that are installed in virtualized operating systems. They provide a cost effective method to implement a server that can monitor many devices, that can easily grow if the number the devices increases.

For the monitored device implementation, as the price, size and power consumption of Android devices has continued to fall while the performance has increased, they are ideal candidates for smart device implementations[2].

This paper presents the design considerations and the implementation problems for a system monitoring a large number of devices, based on Message Queuing technique. The system is designed to lose as little data as possible while keeping communication costs down.

2. THE DESIGN OF THE COMMUNICATION ARCHITECTURE

A simple way to design a server that has IoT devices connected is a standard server-client architecture. Fig. 1 a. In order to reduce the communication costs, UDP can be used to transfer data to the server. The advantage is a reduced communication overhead and the disadvantage is that there is no connection and lost datagrams mush be handled at the application level.

In the following examples, the following keywords will be used: The Server is a windows service that handles two-way communication to the IoT devices. It communicates UDP datagrams to a VPN interface that is connected to the cellular network. The Relay is a service that receives the UDP datagrams, stores them and sends them when they are requested. The relay uses a message queuing server. Rabbit MQ was used for the tests. The Device is the IoT device powered by Android that needs to send data to the server and receive information (such as commands or interrogations) from the Server.

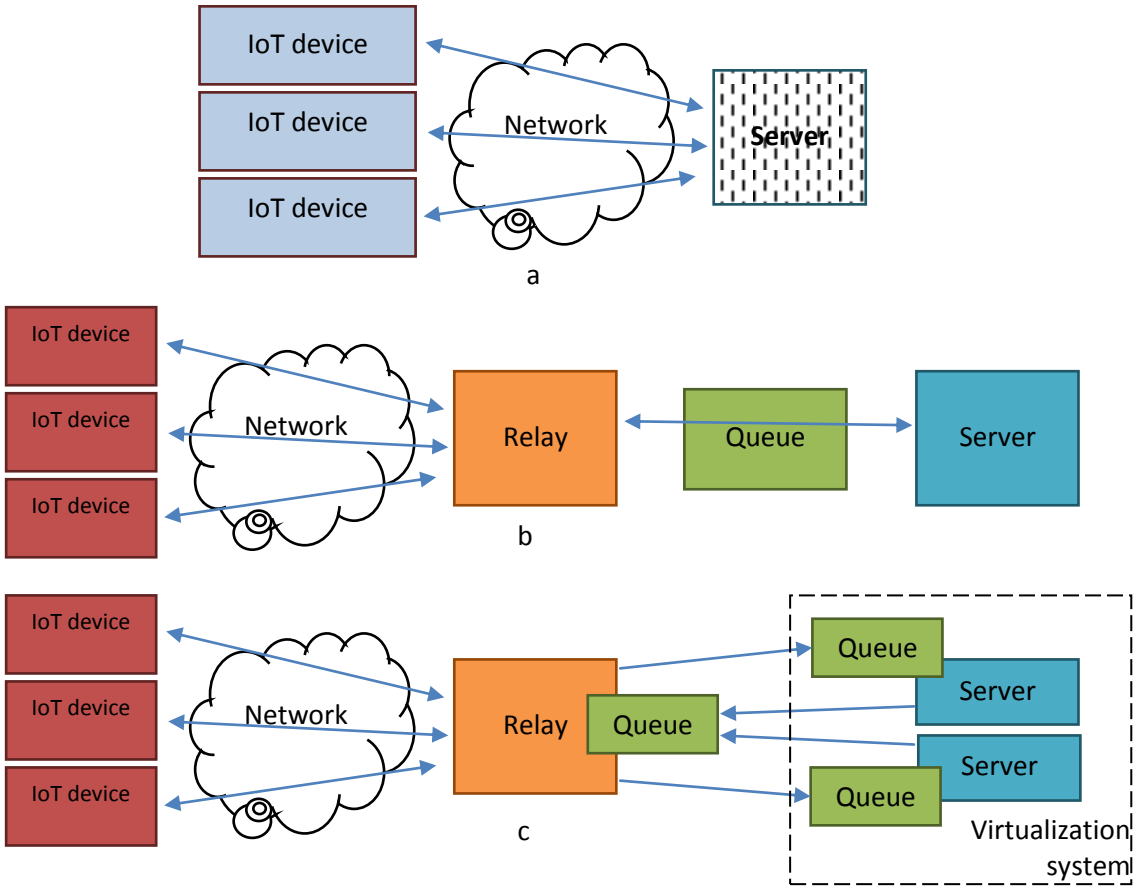

Fig. 1 Evolving the design form a simple server-client architecture (a) to a server side Message Queuing system (b) and finally double Message Queuing system that handles well server and client availability (c).

There are a few problems with this design:

a. If the server crashes or goes down for maintenance, the incoming UDP datagrams will be lost because UDP is unreliable.

b. If the server handles communication and processing, there is no good scalability.

In order to fix this, we can add a "relay" service between them. This can use a Message Queuing server such as Rabbit MQ or Active MQ. This can be seen in Fig. 1. b. In the resulting architecture we have:

a. The Relay that handles the low-level communication: sending and receiving UDP datagrams.

b. The Queue that handles persistence, reliability, in-order processing, etc.

c. The Server that handles high-level communication (parses packets, etc).

In this architecture one server service is present. Usually changes to services are tested on a different server then the one being used in production. In this case the development and production servers run on different physical or virtual machines and need an architecture that handles reliability on both sides: for messages originating from the server and for messages originating from the client.

This setup is also good for scalability if for example, the data needs to be processed before it is stored (such as image processing for form recognition if the devices send images) and more servers are necessary to handle the data. In this third architecture (Fig. 1. c.):

a. Each server has an input queue, for relay-to-server UDP datagrams. This is easier than having multiple readers on the same queue.
b. The relay has an input queue, for server-to-relay UDP datagrams.

For this architecture, the flow of a UDP datagram is the following:

i. The relay listens for incoming UDP datagrams (on all interfaces, on a specific port).
ii. The relay performs a basic check on the packet to confirm that it's the correct packet and it is destined for this server.

iii. The relay envelopes the datagram with extra information (timestamp plus IP/port of the device who sent the message) and then inserts it into the queue for the right server.
iv. The server receives a message in its queue.
v. The server processes the message and performs whatever actions are necessary.
vi. After the message has been processed reliably, it is removed from the queue.
vii. If the server needs to respond to a device, it inserts a message into the relay's queue.
viii. The relay receives a message in its queue.
ix. The relay parses the message's destination and payload, and sends it to the right device.

3. IMPLEMENTATION AND RESULTS

The implementation uses Microsoft Hyper-V Server 2012 Free for implementing the virtualization system, and Windows 8 was used as the OS for the Virtual Machines, running IIS as the server. The data received was written to a SQL server. The server was a quad core, 8-thread, i7, 2670MQ device with 8GB of RAM.

The devices were 25 Android devices running Android 4.2, with dual core processors. The proposed architecture is based on cellular communication as wireless communication was a lower coverage area. Multiple GSM networks in Romania were used in the testing process. The connection speed varied from HSDPA to UMTS.

C# was used to write the code server side and Java for the Android devices.

The message queuing server was RabbitMQ. Java 1.7 was used for the implementation.

In order to secure the data, OpenVPN is used as a VPN client that carries the traffic between the server and the cellular network.

Cellular-based communication used UDP (because it is cost-effective). Messages were binary encoded to reduce the size of the messages.

The purpose of the implementation was for the server to receive all time stamped location information from the devices and be able to send commands to the devices. If the device has connectivity to the server, the device sends the data. If there is no connectivity, the device stores the data locally in order to send it later. The server is also able to send commands to the device. If the device is not in range, the commands are stored in the message queue on the server and sent to the device when it registers to the server.

In order to help the testing, a keep alive mechanism was implemented where a device sends a message to the server even if it has no data to send every 15 minutes. This allows the server to know where the device lost connectivity. A monitoring application was also designed to draw a map with the devices location and to be able to send commands to the devices (Fig. 2).

A first problem of the implementation was caused the unreliable nature of the UDP protocol. Many messages arrived as duplicates because the device determined that, after sending a part of the data that the connection was unreliable and it needed to re-send the data at a later time. Some of the data however arrived at the server. When the device re-sent the data, duplicates occurred. The solution was to check the message pool for duplicates. If the message id was already received, it was marked and logged as a duplicate and was discarded.

A different test was made server side to simulate a server down event, where the server was stopped when devices were sending data. The devices determined correctly that the server did not respond but when the server came back up, a different problem was present.

As the solution for implementing the DB operations was using Microsoft Entity Framework, all database operations were processed and committed one by one (in order to avoid problems with reading incorrect data from the UDP package). If a large number of packets was received in a short period of time, the DB did not have time to properly close all the connections, the connection pool ran out of connections leading to the reject of all new DB connections. A fast solution was to run an SQL agent that tests the number of connections, and terminate them is the number exceeded a certain limit. SQL Server 2014 that was used allows a maximum of 32767 connections by default and has about 5-10 connections on average[3].

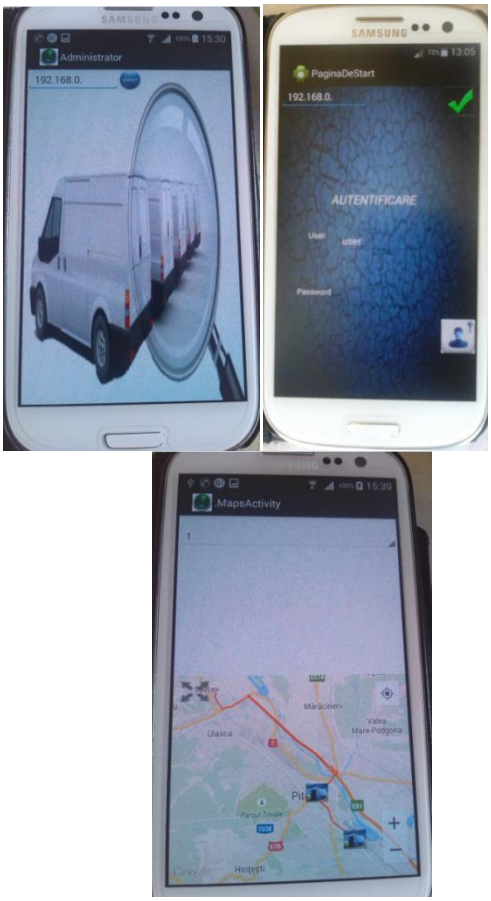

Fig. 2 The Android application allows monitoring the data sent by the other devices using Google Maps.

The SQL script used the following was used to DB terminate the connections (the DB name is 'TestDb'):

```
begin
SELECT @connectionNo= COUNT(*) FROM
master.dbo.syslockinfo WHERE DB_NAME(rsc_dbid) = 'TestDb'
end print @connectionNo

if @connectionNo > 4000
select @query=coalesce(@query,',' )+'kill '+convert(varchar,
spid)+ '; '
from master..sysprocesses where dbid=db_id(@databasename)

if @connectionNo < 4000
begin print 'connection limit ok' end
else if len(@query) > 0
begin
print @query
        exec(@query)
end
```

This solution was not acceptable as positions were lost when the connections were terminated. The final solution was to use the Unit of Work pattern that combines a set of interactions and commit them at once using a transaction[4].

One other problem of the implementation was the fake GSM connectivity. This happens when a device has GSM connectivity but the signal is so low that it is unable to send the data correctly. The solution was to poll the connectivity every minute, with a message/confirmation. If the test message gets the confirmation the connectivity is confirm and the transmission can start correctly.

4. CONCLUSIONS

In this paper an architecture was presented for monitoring a fleet of Android devices that communicate using and unreliable protocol UDP for data transfers. And an implementation was made that showed several practical problems. The implementation problems occurred usually when a large number of devices tried to write a large amount of data to the server, and solutions were found to prevent the server from exhausting the connection pool.

After solving the problem the system monitored 25 Android devices with minimal data loss and connection overhead.

The fact that the problems took place then the single point of failure was turned off (Rabbit MQ on the server) showed that its use for a solution with more might prove limited and further redundancy measures should be investigated.

5. REFERENCES

[1]. Friedemann Mattern, Christian Floerkemeier "From the Internet of Computers to the Internet of Things", Accessed 12.11.2014, Available at: https://technet.microsoft.com/en-us/library/ms187030.aspx

[2] Cognizant reports, "Reaping the Benefits of the Internet of Things", May 2014, Accessed 12.11.2014, Available at: http://www.cognizant.com/InsightsWhitepapers/Reaping-the-Benefits-of-the-Internet-of-Things.pdf

[3]Microsoft, "Configure the user connections Server Configuration Option", Accessed 12.11.2014, Available at: https://technet.microsoft.com/en-us/library/ms187030.aspx

[4] Tom Dykstra, Implementing the Repository and Unit of Work Patterns in an ASP.NET MVC Application, July 30, 2013, ", Accessed 12.11.2014, Available at: http://www.asp.net/mvc/overview/older-versions/getting-started-with-ef-5-using-mvc-4/implementing-the-repository-and-unit-of-work-patterns-in-an-asp-net-mvc-application