# C:\Users\bxs35\AppData\Local\Microsoft\Windows\INetCache\Content.Word\MMC System.jpg4. Implementation

**Figure 4.1 System Configuration**

To test the offloading algorithm in Chaper 2, we build an Offloading System by using the OSGi Framework. We insert the OSGi into the mobile devices and servers to unify the platform, so the both server and mobile will run modules (bundles) in same standard, although they are under different operation systems. There are many kinds of OSGi implements, in which we use the Apache Felix to do our research.

According to Figure 4.1, the experiment system is consisting of three parts: Server Felix, Android Felix and Examination Android Application. The Android Felix is the kernel of the whole system which can provide OSGi services and open APIs for Android applications. When Android OSGi applications are installed, they will connect to the Android Felix, it puts the application bundles into the Bundle context of Android Felix. Then some heavy computation bundles will be uploaded to the Felix environment in server and be remote invoked by Android Felix to realize the offloading algorithm. Finally, the application will be launched and be executed according to the Algorithm 1 in chapter 3.2.2. The following sub-chapters will show the details of implementation of the system.

## 4.1 Felix Transplant

OSGi framework is based on the Java Virtual Machine(JVM), but instead of the JVM, Google develops a special virtual machine called Dalvik Virtual Machine(DVM) for Android OS. DVM is designed for the low memory environments and running multi virtual instances on Android at the same time. Like the Java bytecode can run on the JVM, Dalvik bytecode is a special kind of code which could only be ran on DVM. That is to say, Felix can’t run directly on Android. So the first work is transplanting the Felix into Android OS to build an environment for OSGi modules (Bundle), and we call it Android Felix (or AFelix).

Dx tool in the Android SDK is used to transform the Java bytecode into Dalvik bytecode, the Felix framework bundles need such process to be recognized by Android. This is the transform code:

dex –dx –output=myclasses.dex

aapt add myjar.jar myclasses.dex

Modify an important property of Felix is also necessary before running on Android. Some of the classes in Felix depend on a special Classloader which not exist in Android library. So the method to solve such problem is to change the “org.osgi.framework.bundle.parent” property to the ext to make it invoke the extension Classloader which have been implemented in Android. Now the Felix can be run properly in Android and bundles can also be install and run in the framework.

## 4.2 Android Felix Services

Although bundles successfully run in the Android Felix after the transplant, it’s impossible for the other Android applications to get access to Felix. If we want to do so, we have to repetitively transplant Felix, which is very inconvenient and waste of resources. So we import the AIDL (Android Interface Definition Language) into Android Felix to change it from a normal Android OSGi Application into an Android OSGi Service, then it can provide lots of OSGi APIs for any Android Applications in the Android OS.

### 4.2.1 AIDL Interfaces & Implements

We create AIDL interface file, add it into an Android project and implementing the interfaces make the project become an Android Service.

Refer to the [1], we design the following functions:

* startFelix: Launching the main Felix framework.
* stopFelix: Closing Felix when stop the service.
* installBundle: Installing a specific bundle which was assigned by users.
* uninstallBundle: Uninstalling a bundle which has been installed in the bundle context.
* startBundle: Starting a bundle which has been installed in the bundle context.
* stopBundle: Stopping a bundle which has been installed in the bundle context.
* restartBundle: Restarting a bundle which has been installed in the bundle context.
* updateBundle: Updating a bundle which has been installed in the bundle context.
* sendBundle: Sending a bundle to server.
* executeBundle: Starting a bundle and executing functions in the bundle.
* dependency: Getting the dependency information of a specific bundle.
* networkSpeed: Getting the real-time network speed.
* interpret: Executing some special commands, such as install, uninstall, search, etc.

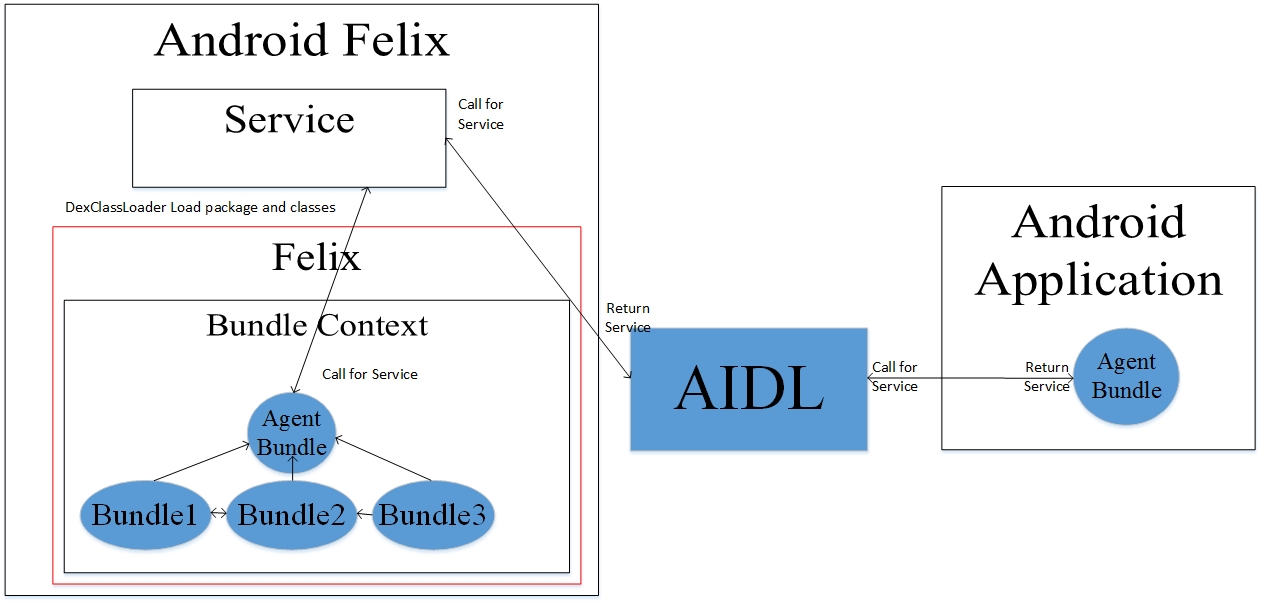
Then we implement these interfaces in the Android Felix. The implements of the interfaces can be divided into three parts:

1. Felix related functions, which are based on the original Felix framework, such as startFelix, startBundle, etc. We import the Felix library as the low-level dependency of Android Felix. Although few APIs of Felix library could be directly used, most methods can’t be invoked directly. So in Android Felix, these methods are override or overwrite in the implements of interfaces. Most of methods of Felix can be realized by just one statement, which is really a good way for application developers to access to Felix without think of the details of Felix APIs.
2. Service related functions, such as executeBundle. This help Android application to get service of bundles. Getting service is the only way to get classes and functions which are encapsulated inside the bundle, but Android can’t recognize the OSGi service. An *agent bundle* system is created for helping Android using the bundles by indirectly getting services.
3. Network related functions, these functions can help Android applications to get network properties, which is an important parameter to the offloading algorithm. And help apps to do bundle transportation between server and mobile devices.

The first part is a pure transplant of Felix, we just implement the same function as the original Felix. The 2 and 3 parts are the extensions of framework, these two parts are designed for compatibility and offloading algorithm.

### 4.2.2 Android Application and Bundle Connection

The OSGi services are implemented by Java Dynamic Classloader technique, getting service is loading the bundle jar package. So it’s impossible for Android applications to static load bundle jars outside the Android Felix.



**Figure 4.2 Android Application Get OSGi Service**

As shown in the Figure 4.2, A special part in AFelix using DexClassLoader (A special Classloader using in Android) to simulate the OSGi service. When applications invoke it by AIDL, the Agent Bundle jar packages are loaded in AFelix Service. Then Agent Bundle return the requesting service to AFelix, and return to applications.

### 4.2.3 File Round-Trip Time Measurement

The offloading algorithm needs the file round-trip transmission time(RTT) in maximum network speed to find the optimal strategy of offloading works to server. However, if no big files were transported by network, it’s impossible to get the RTT. So the AFelix send a file (about 1MB) to the server and measure the time of RTT. The final transmission time, averageRTT will depend on the network statement of the time data measured before. When the time is noted, a new round of file transmission will start to get new transmission time of the next few seconds. And we also limit time of test file transmission to 60 seconds to avoid transmitting too long when the network state is really bad. In this situation, we predict the time by the data transmission amount in the last 60 seconds. This step will start when the framework is launched and will temporarily stop when the program starts to execute or some other applications are using the network to avoid choking the network channel. The other problem is we will never know the real transmission time. The RTT return to applications are the average speed in the last a few second, but not the real-time. Refer to the paper [2], the following formula was used to compute the RTT of the test file:

By repetitively doing this step, the AFelix will get an approximately RTT time for using in offloading algorithm.

## 4.3 Remote Service

We install Felix on server and build the remote connection between mobile devices and server by R-OSGi. R-OSGi is also a bundle, it uses ASM library to create agent bundles (this is totally different with the bundle we used in *Agent Bundle System*) for local bundles and these local bundles can connect with the bundles on the server by agents. We change R-OSGi into the Dalvik bytecode and run it into Server Felix and Android Felix. However, the agent bundle is created in Java bytecode and it’s created automatically during the remote invoke process. So refer to the paper[3], we import the Dex service (Dex is the technique to transform the Java bytecode to Dalvik bytecode) into the service. This step makes the Felix framework can automatically transform bundles which installed in it from Java bytecode into Dalvik bytecode. The transfer codes are below:

File f=File.createTempFile(” temjar”,”.jar”, new File(” //data//local//tmp”));

f.createNewFile();

FileOutputStream fos = new FileOutputStream(f);

fos.write(bytesfromjar);

fos.flush() ;

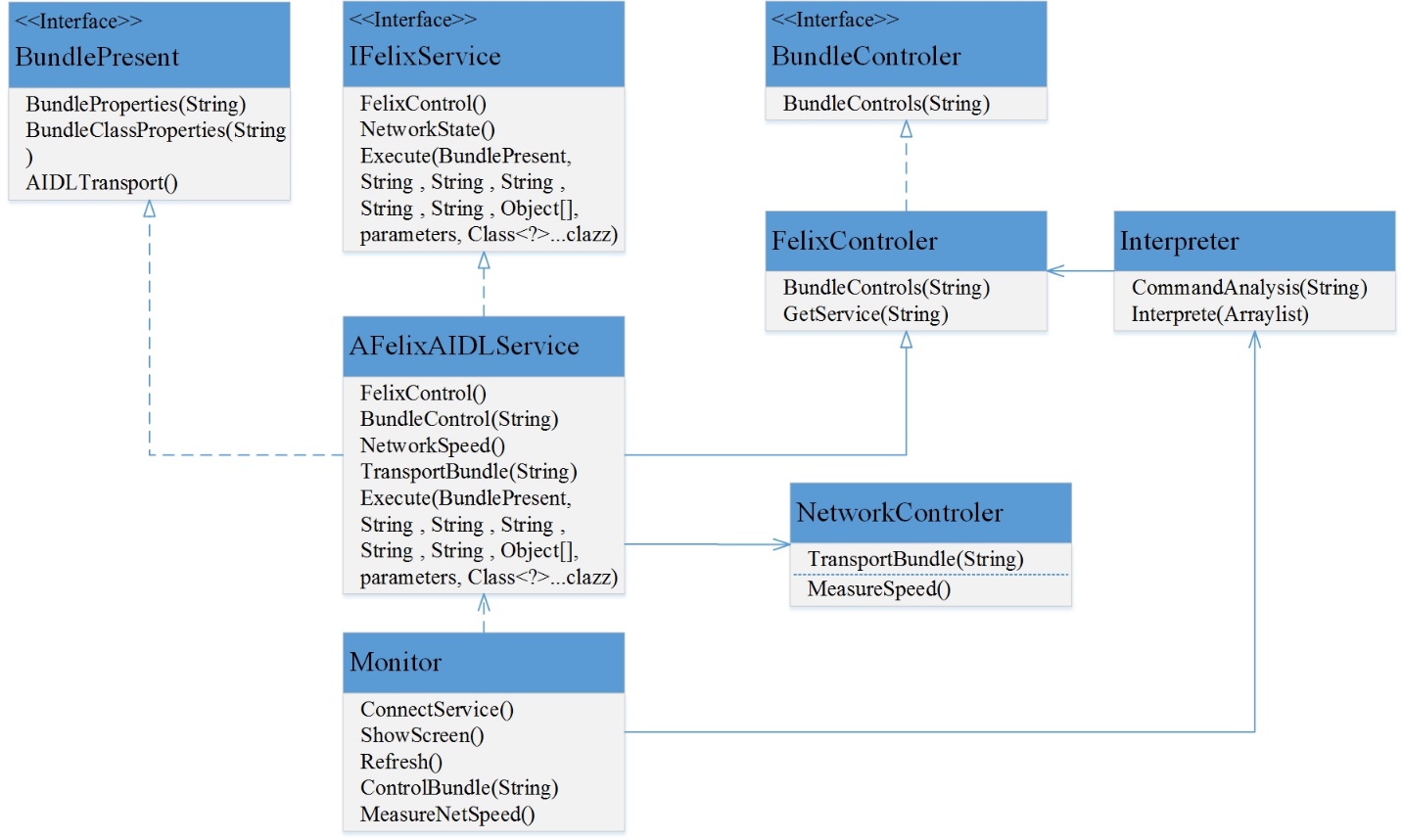
fos.close() ;

DexService dexService = new DexServiceImpl();

byte [] dexJar = dexService.createDexedJarFromFile(f.getAbsolutePath());

return new ByteArrayInputStream(dexJar);

## 4.4 More details

According to the offloading experiment demand and develop scheme in 4.1-4.3 we design the following architecture for Android Felix:

**Figure 4.3 Android Felix Class Diagram**

To provide general service, the IFelixService AIDL interface was opened to applications. AFelixAIDLService is the kernel of AFelix which assemble the Felix and network components to implement the AIDL interface. In these components, the FelixControler overwrite and encapsulate Felix functions. It also realizes the function to get OSGi service for Android Application which mentioned in 4.2.2. While the another component NetworkControler focuses on the network functions mentioned in 4.2.3.

Considering during the experiment we need to monitoring Bundles and network statements. The following monitor in Figure 4.4 was designed. As shown in the Figure 4.3, the monitor has connected to the Android Felix service, so it could get the statement of bundles and network speed to show them on the screen. It also includes an interpreter and can execute some advanced functions such as install bundles from anywhere in the file system and test to get service from a specific bundle.

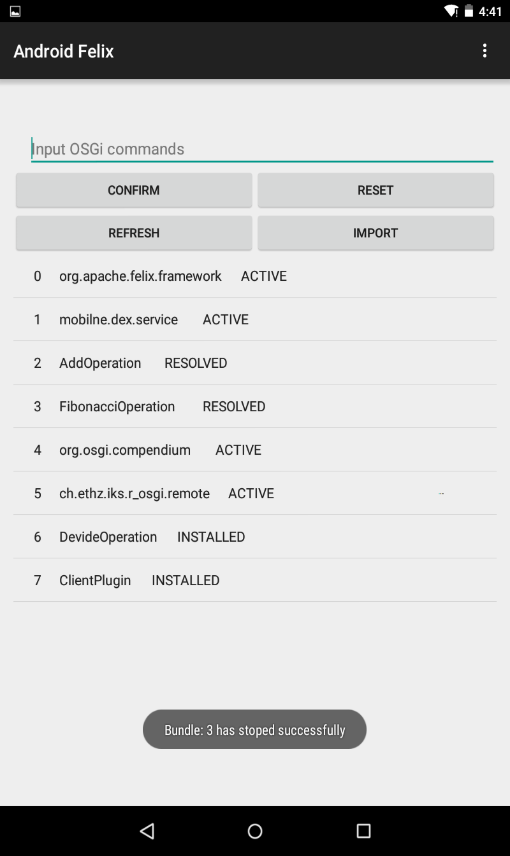
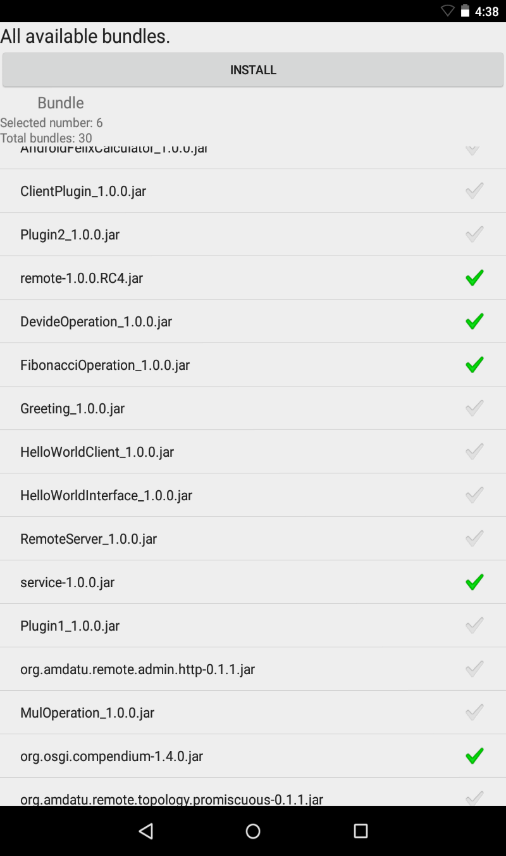
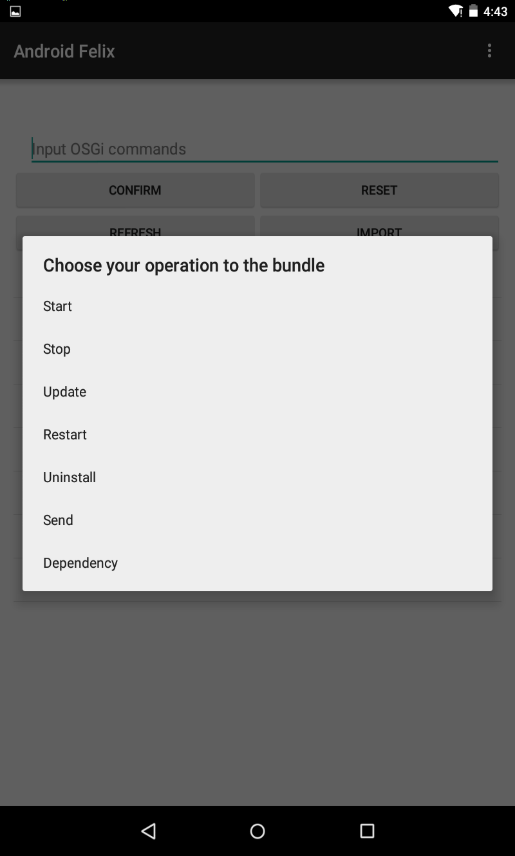


Figure 4.4(a) Main Control Interface

Figure 4.4(c) Bundle Control

Figure 4.4(b) Bundle List

## 4.5 Ready for Experiments

Until now, Android OSGi system has been successfully built, we built some OSGi software to simulate some typical software. They are used to do some experiments to compare our optimized offloading algorithm with the local computing and traditional full-offloading computing. The experiments data are filled in some tables then made some diagrams. We analyze the data by these tables and diagrams to get to our conclusion. The following 2 chapters will show the details of our experiments, data analysis and the conclusion.

# 5 Experiments

To verify our offloading algorithm, the three major verities, computing amount, data transmission amount between computing modules and network speed will be tested and show the influence to the offloading strategy and computing speed when one of them are changed while the other two elements remain unchanged. We design two typical software modules for each software pattern, chain and tree. One module is the *Scientific Computing Module* (SCM), which has medium data transmission between bundles but low computing amount at beginning and low data transmission between bundles but high computing amount at end. The another module is the *Graphic Dispose Module* (GDM), which has high data transportation and medium computing during the whole procedure. The major element of SCM is the computing amount, so it was used to test the influence of computing amount. And GDM focus on data transmission, so we do experiments related to data transmission amount between computing modules and network speed. We also compare the computing time of our offloading algorithm to full local computing and traditional full offloading computing. These comparisons will be showed in tables and diagrams.

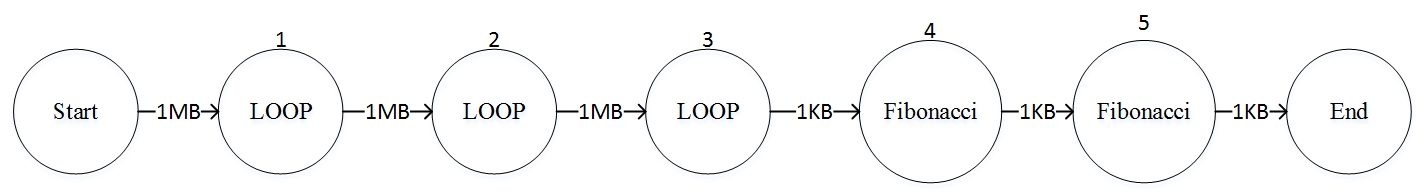
All of our experiment environment are executed on Google Nexus 7 as the mobile device and Lenovo laptop as the server.

## 5.1 Chain Pattern

In chain pattern, all computing modules are executed step by step from the beginning to the end. Our experiments use 5 bundles include 3 loop bundles and 2 Fibonacci bundles to test the computing speed. The loop bundles are used to do fine turning and the Fibonacci bundles will create huge speed interval. We don’t change the bundle when start experiments to avoid the internal disturb. Instead, the bundle order will be changed to make the SCM & GDM.

### 5.1.1 SCM

We design the SCM as shown in the following figure:



**Figure 5.1 Chain SCM**

The computing amount order is M(1) < M(2) < M(3) < M(4) < M(5), the data transmission and the network statement were remained unchanged. We rise the compute amount of every bundle to rise the total compute amount every time. The range of total compute amount of this experiment is from 10^8 to 2\*10^13.

The experiment result is shown as the below chart and diagram:

**Table 5.1 Chain SCM Experiment Result**

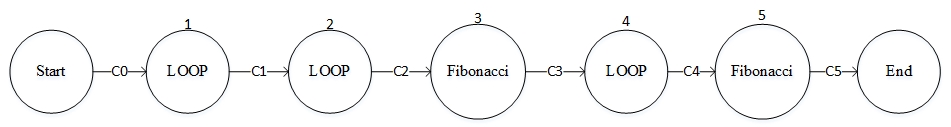
|  |  |  |  |
| --- | --- | --- | --- |
| Computing Amount(10^9) | Local(ms) | Full Offloading(ms) | Optimized Offloading(ms) |
| 0.1 | 152 | 2182 | 171 |
| 0.5 | 203 | 1751 | 206 |
| 1 | 301 | 1873 | 259 |
| 10 | 1791 | 1872 | 443 |
| 100 | 4474 | 3484 | 936 |
| 500 | 6480 | 3182 | 2089 |
| 1000 | 8253 | 3465 | 1097 |
| 5000 | 26928 | 5241 | 2828 |
| 20000 | 69554 | 9446 | 7329 |

**Figure 5.2 Chain SCM Experiment Result Line Diagram**

It’s clear that with the increasing of calculating amount, the time of calculating exponential increase, while the two offloading strategies linear increase. The full offloading cost long time when the computing amount is small, while the optimized offloading remain unchanged and similar to the local before 10^9 computing amount. The optimized offloading shows significant advantage between 10^9 to 10^11, especially around 10^10 which accelerate about 75% than the faster strategy in the other two offloading strategy. When the computing amount become larger than about 10^11, both offloading strategy begin to faster than local. And the optimized offloading increases gradually and always faster from 22% to 73% than full offloading. However, the advantage of optimized is weaken when in tremendous computing situations.

### 5.1.2 GDM

We design the GDM as shown in the following figure:



**Figure 5.3 Chain GDM**

The total computing amount is fixed to 10^11, and the computing amount in every bundle are remained unchanged. The data transmission increase from 10MB to 80MB, and all transmission data amount between bundles change at the same time with same amount between different experiments. For example, C0-C5 are all 1MB in the first experiment and if the C0 doubles in the second experiment, all other C1-C5 will become 2MB.

To put pressure on the full offloading and optimized offloading, data transmission order is: C0 = C2 = C4 > C1 = C3. We choose 4 typical situations of data transmission amount: 10, 20, 40 and 80MB to do experiments. And in every situation, we also test the influence of network speed by limiting the max data transmission speed between local and server. The result of these experiments are shown in below table and diagrams:

**Table 5.2 Chain GDM Experiment Result**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Data Amount(MB) | Local | Full Offloading | Optimize Offloading | Network Speed | Offloading NO.  (-1 is local computing) | Network Statement |
| 10 | 14475 | 3585 | 3623 | 2208.3 | 2 | Normal Network |
|  | 14376 | 6557 |  | 3 | Slow Network (200-300KB/s) |
|  | N/A | 14421 |  | -1 | Worst Network(<50KB) |
| 20 | 14731 | 5436 | 4038 | 2221.7 | 2 | (same as 10 MB) |
|  | 42570 | 11990 |  | 3 |  |
|  | N/A | 14481 |  | -1 |  |
| 40 | 14932 | 9103 | 4090 | 1955.1 | 3 | (same as 10 MB) |
|  | N/A | 14806 |  | -1 |  |
|  | N/A | 14723 |  | -1 |  |
| 80 | 15170 | 15881 | 4936 | 2182.2 | 5 | (same as 10 MB) |
|  | N/A | 14770 |  | -1 |  |
|  | N/A | 14770 |  | -1 |  |

**Figure 5.4 Computing time of 3 offloading strategy in normal network statement**

**Figure 5.5 Computing time of optimized offloading**

The Table 5.2 and Figure 5.4 shows the full offloading perform good in the low data transmission situation, even better than the optimized offloading. With the increase of data transmission amount, the performance of full offloading drops quickly and when the data amount become really large, the full offloading strategy will slower than local strategy. The optimized offloading keep stable rather than traditional full offloading, because the computing time will not increase dramatically with the increase of data transmission amount. So the optimized offloading show superiority than the full offloading in big data situations of the chain pattern.

Table 5.2 and Figure 5.5 shows us the influence from network statement or the network speed to optimized offloading algorithm in chain pattern. We could see from the Table 5.2 that when the speed of network is manually limited, the testing program which using the optimized offloading will dynamically choose the bundle which is best for shorten computing time. That is to say when the network statement becomes slower, the algorithm will find if the original strategy will still be the best of offloading and decide to change the strategy or not. And when the network statement is really bad, in most situations the testing program will choose to execute on mobile devices otherwise the data transmission amount is really small.

## 5.2 Tree Pattern

### 5.2.1 SCM

### 5.2.2 GDM

Reference

[1] Bouzefrane, Samia, Dijiang Huang, and Pierre Paradinas. "An OSGi-based Service Oriented Architecture for Android Software Development Platforms."*Proceedings of 23rd International Conf. on Systems and Software Engineering and their Applications (ICSSEA’2011)*. 2011.

[2] Paxson, V., M. Allman, and Computing TCP’S. Retransmission Timer. "RFC 2988." *Computing TCP's Retransmission Timer* (2000).

[3] Zhang, WeiShan, et al. "An OSGi-based flexible and adaptive pervasive cloud infrastructure." *Science China Information Sciences* 57.3 (2014): 1-11.