Network Performance Isolation Scheme for QoE in a mobile device

Woonghee Lee*, Hyunsoon Kim*, Joon Yeop Lee*, Albert Yongjoon Chung*, Yong Seok Park†, and Hwangnam Kim*
*School of Electrical Engineering, Korea University, Seoul, Korea
†DMC R&D Center, Samsung Electronics
Email: *{tgorevenge, gustns2010, charon7, aychung, hnkim}@korea.ac.kr and †yongseok.park@samsung.com

***Abstract*—Today's smartphones and tablet PCs are equipped with various sensor units and wireless capabilities along with high performance computing units, which make these mobile devices capable of providing a wide range of services. Because of such characteristics, mobile devices consume more network traffic and frequently use multiple Internet services at the same time. The performance degradation of a foreground service happens frequently due to concurrently running background services in a mobile device. In this paper, we propose *NetPIS*, Network Performance Isolation Scheme for QoE, which resolves the aforementioned problem by applying the concept of performance isolation to the foreground service. *NetPIS* is the receiver-based scheme suitable for mobile devices without any modification on senders. Furthermore, unlike most performance isolation implemented by virtual machines, we suggest a scheme that does not require virtualization that might be heavy for mobile devices. The proposed scheme was implemented on a smartphone by modifying the kernel, and various experiments were conducted to evaluate the advanced system behavior of *NetPIS*.**

***Keywords*—Network performance isolation, QoE, Mobile device**

I. Introduction

There are various applications that utilize the Internet services on mobile device, and network traffic incurred by these applications has been growing rapidly. In fact, the Ericsson mobility report [1] predicts that the amount of traffic per mobile device subscription will increase five-fold between 2013 and 2019. Moreover, smartphones and tablet PCs mostly provide multi-processing functionality, which can generate concurrent Internet services. To test the frequency of concurrent Internet services usage, we collected traffic statistics of 6 users' devices 24 hours long using an application we designed. As a result, the time of the concurrent Internet services running on a device differs for each user but this time reaches up to about 34% of the total. Furthermore, 92.3% of the total traffic was receiving traffic, which means that the downlink traffic is dominant in mobile devices.

Without mitigating the interference between foreground and background services on the concurrent wireless communications in one device, Internet services may meddle in one another, resulting in starvation on the service that might be important to the device user. For instance, file download or application update services on a mobile device can downgrade the performance of a foreground service such as video streaming or Internet browser, and this problem is distinctly shown in the video we made [2]. Without a sophisticated solution for the interference mitigation, this effect can severely affect the quality of experience (QoE) of the user who directly interacts with a foreground service. Therefore, performance isolation to mitigate the interference between foreground and background services is required with priority. We designed our system in a way to satisfy the aforementioned requirement, which will give better QoE to users while sustaining resources to concurrently running background processes with certain level. Instead of the traditional performance isolation, our goal is to provide enough resource to the foreground process, and give the rest of the resource to the concurrently running background processes without loss of the overall network resource.

Network performance isolation is important in various areas and can be divided into two types [3]: virtualization and flow level solution. Network performance isolation through virtual machines is advantageous in the aspect of easy isolation without the sensitive control of the network parameters. Indeed, most of the performance isolation approaches in various areas are mainly done by virtualization which restricts the amount of allocated resource per virtual machine. However, the aforementioned network virtualization approach is not actively researched in the area of mobile devices. This is due to the limited hardware performance of mobile devices compared to that of large scale systems, which means the processing overhead of a virtual machine can be greater than its advantages [4]. Enabling virtual machine for mobile devices may enlarge the delay of inter-process communication and eventually lead to lower battery lifetime. This is why the flow level isolation is crucial in the area of mobile devices, although deep understanding of wireless network and congestion control methods from various protocols is required. However, even the flow level approach does not actively support the isolation in the aspect of a data receiver. This is because most of the flow controls are based on senders rather than receivers, resulting in the limited control in the aspect of receivers. In reality, such solutions are hard to be applied to mobile devices because most mobile devices use Internet services that usually produce downlink traffic rather than uplink traffic.

Aforementioned problems can be categorized into two major challenges in designing network performance isolation scheme for mobile devices. Firstly, the hardware performance of a traditional mobile device is still limited, which means a virtualization based solution is not appropriate yet. Secondly, most of the mobile devices usually receive data rather than transmit data, which makes currently proposed flow based approaches difficult to be implemented in mobile devices. In order to overcome these challenges, we propose a Network Performance Isolation Scheme (*NetPIS*), and the effect of *NetPIS* is shown in a demonstration video we made [5]. *NetPIS* adaptively controls several Transmission Control Protocol (TCP) parameters to provide adequate level of network

978-1-4673-8590-9/15/$31.00 ©2015 IEEE

bandwidth isolation for a foreground service when background services run concurrently. Among transport layer protocols, we focus on TCP because it is the most widely used transport layer protocol, and more than 95% of total data traffic over the Internet is transferred over TCP [6].

The contribution of *NetPIS* to current mobile devices is listed as follows. (*i*) *NetPIS* guarantees higher service quality for a foreground service and maintains the overall performance of a device. (*ii*) *NetPIS* is the receiver-based scheme suitable for mobile devices. (*iii*) *NetPIS* provides a flow level control without using any virtual machine, which can be a burden on mobile devices. Thus, a *NetPIS*-enabled device consumes similar amount of power to a unmodified device. (*iv*) Proposed isolation technique adjusts the receive buffer size efficiently, which provides more usable memory for a mobile device.

In order to prove the applicability of the proposed scheme, we implemented *NetPIS* on the kernel space of a real smartphone named Galaxy Nexus, and multiple evaluation studies were conducted to show the distinctive performance of *NetPIS*.

This paper is organized as follows. Section II shows prior researches that have similar purposes or techniques compared to *NetPIS*. Section III lists control parameters used by *NetPIS* and justifies the reasons of using these parameters. Section IV describes the overall design of *NetPIS* and the implementation architecture made in kernel space of Android mobile platform. Section V explains evaluation environments and results that prove the performance enhancement made by *NetPIS*. Finally, Section VI concludes the paper.

II. Related Work

There are various performance isolation researches targeting the mitigation of interference among multiple processes (computing units generated by Internet services) running on a same device or system. Especially in data centers, isolating performance among different tenants is a crucial issue [7]. Since the level of service should be proportional to what the tenants expect out of their payment, the performance downgrade of a tenant due to heavy traffic of other tenants is never tolerable. Therefore, virtualization is often applied to support performance isolation in the perspective of the network data rate in data centers.

Other than aforementioned large scale systems such as data centers, there are also several researches that tried to perform performance isolation on individual devices. For instance, [8] suggests flexible virtualization which virtualizes specific layers only in order to reduce the overhead from the layers that are unnecessarily virtualized. In addition, SR-IOV system [9] is a generic virtualization architecture that enhances CPU utilization with proper isolation of CPU resource among processes.

Among many researches focusing on performance isolation, some researches considered user interactions with devices in order to improve QoE. Y. Etsion *et al.* proposed a process prioritization scheme that concentrates CPU resources on user-interacting processes for processing performance isolation [10]. Similarly, H. Zheng *et al.* suggested a processor scheduling framework to guarantee the performance of user-interacting processes [11]. However, these researches focused on improving performance in terms of only processing without considering network performance. Therefore, these researches cannot be used to guarantee the network performance of application with which an user interacts.

Other than isolation focused on the processing resource such as CPU usage, there are few researches focused on network performance isolation within a mobile device. In addition, virtualization is not highly recommended because most virtual machines (VMs) are heavy to low performance devices such as smartphones. In fact, Android mobile platform recently discarded Dalvik VM. Therefore, most network performance isolation schemes for mobile devices use flow level control. Among the researches on flow level control, some researches focused on modifying TCP parameters to dynamically assign resources to multiple ongoing flows. Some researches [12], [13] tried to adjust both sender and receive buffer sizes to increase or decrease the TCP throughput. However, these systems require modification on both the data sender and the receiver, which is not easily applicable to the real devices that usually receive data from different servers of various Internet services. In order to solve such the scalability problem, some researches such as [14], [15] suggested adjustment only on the receiver. However, such an adjustment only lowers the data rate of low priority services, and there is no modification on the high priority services that require more resource allocation.

In order to solve these problems that prior work could not resolve, we propose *NetPIS* suitable for mobile devices, which is designed to isolate performance of a foreground service from interfering flows. Unlike prior related work focusing on the processing performance isolation, *NetPIS* focuses on the network performance isolation to guarantee the QoE of mobile users. *NetPIS* adjusts per-flow TCP parameters in a way that immediately increases data reception rate of a foreground service and downgrades receiving performance of concurrently running background services. Although *NetPIS* increases data rate of a foreground service by sacrificing the quality of concurrently running background services, the scheme does not affect the overall network performance of the device.

III. Control parameters for *NetPIS*

Advertised window is a notification parameter which reports the condition of the receive buffer to the data sender. According to the advertised window size, the sender can adjust the next window size to perform appropriate flow control. *NetPIS* adjusts the advertised window size which belongs to either foreground or background processes. If the advertised window belongs to a foreground process, in order to utilize the full bandwidth of the network, *NetPIS* enlarges the advertised window size and prevents the advertised window from becoming the restriction of sender's window increase. On the other hand, *NetPIS* decreases the advertised window size if the process is a background process. This tendency is similar to most of the prior work we mentioned in Section II, which means solely modifying this parameter cannot stably isolate the network performance of the foreground process.

In TCP, final congestion window size is normally decided by the minimum size between the sender's window and the receiver's advertised window, which means the final window size cannot be increased unless the sender's window size increases along with the advertised window size. Although the window size always increases additively or multiplicatively, the data rate of the foreground process will always be bounded to the congestion window size. In addition, if the background process started first and already took much of the network resource, the foreground process cannot get enough resource.

This constrained performance of the foreground process can be breached by downgrading the performance of con-

currently running background processes, but it is hard to know what degree of window size reduction is optimal. If the advertised window size is smaller than the sender's window size, the performance will be proportional to the advertised window size. However, if the advertised window size is greater than the sender's window size, advertised window size needs to be lowered even more. Therefore, it is hard for the receiver to make such decision since the data receiver cannot know the sender's window size. Too much downgrading of the advertised window can possibly lower the overall performance of the mobile device, and little adjustment sometimes merely affect the performance of the background process. This is why most of the prior work focused on changing both sender and receiver side window variables. However, it is a serious drawback of the aforementioned work that modifications of servers are necessary. In addition, most of the wireless data generated by mobile devices are downlink data. Therefore, the performance isolation on mobile devices should be done in the aspect of a data receiver.

In order to overcome the limitation of prior work, *NetPIS* adjusts additional parameters from delayed ACK mechanism [16]. Delayed ACK mechanism is a TCP technique, which reduces ACK sending overhead to increase efficiency of the network. TCP data receiver was originally designed to acknowledge every packet it received. However, some services, such as HTTP, send very small packets to user devices, leading to frequent ACK transmissions by the users. In order to prevent this inefficiency, delayed ACK mechanism utilizes a value called bounded frame size, which prevents a TCP receiver from sending an ACK until the total payload size reaches the bounded frame size no matter how many packets were received. When filling up the bounded frame size takes too much time, the mechanism uses ACK timeout (ATO) to prevent the sender from unnecessary retransmissions.

However, less ACK transmissions cause the sender's window size to increase slowly. Therefore, using the behavior of this mechanism, we can decrease the data rate of the background process by enlarging the bounded frame size and ATO to certain values. No matter how large or small the congestion window size of the data sender is, enlarging the delayed ACK parameters will slow down TCP data rate. Unlike the advertised window size that has limitation on controlling TCP flow, delayed ACK parameters can directly affect the flow by forcing data sender to slow down transmission. Moreover, less number of ACKs sent by the background processes will eventually provide more chances of data reception for the foreground process, resulting in a faster performance recovery of the foreground process when the background processes already took good spots for communications.

In addition to the above parameters, *NetPIS* also controls receive buffers of both foreground and background processes. We found out that device memory is exceedingly wasted because of the unnecessarily large size of default buffer. In addition, many Internet services open tens of sockets for faster Internet service. Therefore, a large portion of receive buffers is wasted even though most part of the receive buffers are not even used by the service. In order to prevent such waste, *NetPIS* controls the receive buffer size for all processes according to their needs.

IV. *NetPIS* Design and Implementation

In this section, we describe the overall design of *NetPIS*. Also, we give a detailed explanation of *NetPIS*'s major components and implementation.

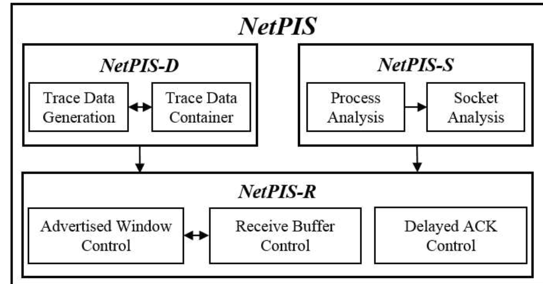


Fig. 1: The overall design of *NetPIS*

A. Overall design

Figure 1 shows the overall design of *NetPIS*. As shown in the figure, *NetPIS* consists of three major components: *Data communication tracer* (*NetPIS-D*), *Socket classifier* (*NetPIS-S*), and *Resource allocator* (*NetPIS-R*). *NetPIS* allocates the different amount of network resources to each process in order to guarantee the isolated performance of foreground process and to improve user QoE as much as possible. In order to allocate resources to processes properly, *NetPIS-D* monitors the data reception of each process and analyzes the data communication characteristics. After that, *NetPIS-D* hands over the analyzed characteristic of each process to *NetPIS-R*. *NetPIS-S* takes a roll of finding the sockets conducting the data transmission for a foreground process and notifying which sockets belong to the foreground process to *NetPIS-R*. Finally, using information provided by *NetPIS-D* and *NetPIS-S*, *NetPIS-R* allocates data communication resources to processes accordingly. In order to implement *NetPIS*, we modified the network stack of Linux-based kernel, so *NetPIS* can be flexibly applied to other systems.

B. NetPIS-D

NetPIS-D takes a role of tracing and analyzing the network usage of each process that is most likely to contain more than one socket. After that, *NetPIS-D* hands the network usage information over to *NetPIS-R*, and *NetPIS-R* appropriately controls the network resource using the information. Therefore, *NetPIS-D* is necessary for an accurate resource control of each process. In kernel, received data is temporarily stored in the receive buffer, and afterwards the process takes some or all of the stored data. If the amount of received data is larger than the amount that the process can take at once, the remaining data is stored in the receive buffer until the process takes it in next time. At this moment, the required size of the receive buffer is twice the remaining data size in order to prevent an overflow of the receive buffer. *NetPIS-D* traces the required size of the receive buffer of each process's sockets continuously, analyzes this traced information, and provides *NetPIS-R* with each process's data communication characteristics such as the average, minimum, and maximum required size of the receive buffer. Therefore, *NetPIS-D* sends feedbacks to *NetPIS-R* over and over, and *NetPIS-R* can control each socket's allocated network resources using latest information.

After *NetPIS-D* finishes generating the trace data, *Trace Data Container* in *NetPIS-D*, as shown in Figure 1, is periodically called to save the overall network trace data. *Trace Data*

Container creates a file that records the trace information to the sdcard of the mobile device, which allows to maintain the trace data safely, regardless of device reboot or system error.

C. NetPIS-S

Basically, end to end communications are conducted by sockets, and sockets contain network related resources. Therefore, *NetPIS-R* controls each socket's allocated network resources. In addition, today's many applications concurrently use multiple sockets rather than only one socket. Thus, in order to provide more network resource to the foreground process, sockets conducting data transmission for the foreground process need to be distinguished from other sockets. Thus, *NetPIS-S* conducts two analyses for socket classification.

1) Process analysis: Process analysis is the first step of socket classification. Process analysis is designed to filter out the on-going foreground process from multiple processes, which can be further categorized into two types: user processes and system processes. User processes are defined as processes which interact with user directly, whereas system processes refer to processes executed by the operating system or the framework mostly. Most foreground or background processes considered in this paper are user processes. Therefore, process analysis only targets the user processes. Basically, user processes are created by the zygote process in Android OS. Thus, *NetPIS-S* can know that a process is an user process if the parent process ID (*ppid*) of the process equals to the process ID (*pid*) of the zygote process. Among the user processes, *NetPIS-S* finds the foreground process containing zero *oom_adj*, because a process's *oom_adj* value is zero only if the process is a foreground process. *oom_adj* is one of the variables found in *proc* file system in kernel and indicates the priority of the process.

2) Socket analysis: After finding the foreground process, it is necessary to find which sockets conduct the data transmission for the foreground process. Therefore, *NetPIS-S* includes an additional step called socket analysis. There are several cases for using sockets, and the most straightforward and general case is when the foreground process makes sockets for itself and uses them. In this case, *NetPIS-S* knows that the socket belongs to the foreground process if pid of a socket owner process equals to the foreground process's pid. The second case is when the foreground process's child process conducts data communications. In this case, the foreground process does not make a socket for data communications. Instead, the foreground process makes a new child process which conducts data communications on behalf of the foreground process. Therefore, *NetPIS-S* knows that the socket belongs to the foreground process if $ppid$ of the socket owner process equals to the foreground process's pid.

Additionally, in some exceptional cases, *NetPIS-S* can distinguish sockets using additional parameters such as *process name*. For instance, *NetPIS* does not limit the network resources of the processes, such as VoIP services, which are frequently used as background processes but should have high priority. After socket classification in terms of foreground or background, *NetPIS-S* hands over the classification result to *NetPIS-R*.

D. NetPIS-R

A role of *NetPIS-R* is to set each socket's receive buffer, advertised window, and delayed ACK parameters properly. *NetPIS-R* uses three algorithms to adjust these parameters. However, the algorithm for receive buffer control is similar to that of advertised window control because the size of the advertised window is derived by using the receive buffer size. Therefore, we will explain the algorithms of receive buffer and advertised window control together. Then the delayed ACK control algorithm will be explained separately. In order to implement the aforementioned algorithms, we modified the network stack of kernel, which makes our algorithms run packet-wise.

1) Algorithms for receive buffer and advertised window control: Basically, we created a base algorithm for the adjustment of receive buffer and advertised window. In addition, a supplementary method is added to the base algorithm in order to cover diverse situations.

Base algorithm: Suppose that f and b provided by *NetPIS-D* indicate the required resource size of the foreground and background processes respectively. In this situation, the data communication of the foreground process is in competition with those of the background processes. Therefore, it is reasonable to allocate resource larger than f to the foreground process in order to guarantee its performance. On the other hand, the amount of resources for the background processes should be limited. By doing so, the connection of the foreground process will be superior to that of the background processes, and the network performance of the foreground process can be isolated. Thus, *NetPIS-R* gives the foreground process's sockets an advantage by multiplying f by $(1 + \alpha)$. α indicates the weight factor which is between zero and one. On the contrary, a weight factor β acts as a disadvantage for the background processes by multiplying b by $(1 - \beta)$. Like α, β is between zero and one.

Algorithm 1 and 2 show the pseudo codes of the receive buffer and the advertised window control respectively. As we have just explained, the receive buffer size is newly decided in terms of the priority of the process in line 2 and 9 of Algorithm 1. $buffer$ in Algorithm 1 indicates the required buffer size of process's socket, and it is provided by *NetPIS-D*. Because *NetPIS-D* provides *NetPIS-R* various statistics such as the average, minimum, and maximum required size of the receive buffer, the used information can be different depending on the purpose. Basically, *NetPIS-R* uses the average required buffer size for $buffer$. As we explained before, *NetPIS-D* consistently traces the required size of the receive buffer of each process's sockets, provides latest network usage information, and sends feedbacks to *NetPIS-R*. Thus, the new value of $buffer$ is used whenever Algorithm 1 runs, and *NetPIS-R* can adjust network resource using the latest network usage information. Note that $netpis_buffer$ does not become the new value of $buffer$ in next iteration because the new value of $buffer$ is newly decided by *NetPIS-D*. Therefore, the size of buffer allocated to each socket cannot increase or decrease infinitely.

In a similar way, the size of advertised window is newly determined in line 5 and 11 of Algorithm 2. win in Algorithm 2 is the size of advertised window, and its derivation follows the original TCP using $buffer$. Therefore, similar to $buffer$, the new value of win is used whenever Algorithm 2 runs. Basically, *NetPIS-R* increases the allocated resources to the foreground process's sockets and limits the network resources of the background processes' sockets to guarantee the performance of the foreground process. However, there is no need to restrict the background processes' resource when the

Algorithm 1 Receive buffer control

```
1: if a socket belongs to the foreground process then
2:     netpis_buffer ← buffer * (1 + α)
3: else
4:     if only background processes exist then
5:         netpis_buffer ← buffer
6:     else if it is in Case 2 then
7:         netpis_buffer ← buffer
8:     else
9:         netpis_buffer ← buffer * (1 − β)
10:    end if
11: end if
```

Algorithm 2 Advertised window control

```
1: if a socket belongs to the foreground process then
2:     if it is in Case 1 then
3:         netpis_win ← win
4:     else
5:         netpis_win ← win * (1 + α)
6:     end if
7: else
8:     if only background processes exist then
9:         netpis_win ← win
10:    else
11:        netpis_win ← win * (1 − β)
12:    end if
13: end if
```

foreground process does not generate any data transmission. In such the case, *NetPIS-R* allocates the required resource to the background processes' sockets without resource restriction. This operation is shown in line 5 of Algorithm 1 and in line 9 of Algorithm 2.

Supplementary method: In order to prevent possible problems caused by the resource allocations, *NetPIS-R* configures the advertised window and the receive buffer size of each socket depending on the circumstances.

Case 1 specified in line 2 of Algorithm 2 is when *NetPIS-R* increases the allocated resources of sockets. In this case, *NetPIS-R* increases the receive buffer size and then the size of advertised window. If *NetPIS-R* does not modify the aforementioned parameters in this order, a server may calculate the size of a new data using the new value of the advertised window and send a data, which is bigger than the amount that the receiver can receive. Consequently, a drop of partial data may occur at the receiver side. Therefore, *NetPIS-R* should prevent such possible loss by increasing the receive buffer size first. In line 3 of Algorithm 2, the size of the advertised window does not increase although the socket belongs to the foreground process, which means that *NetPIS-R* waits for the increase of the receive buffer.

On the contrary, Case 2 in line 6 of Algorithm 1 is when *NetPIS-R* decreases the allocated resources of sockets. In this case, *NetPIS-R* reduces the size of the advertised window and then the receive buffer size. Fundamentally, the server decides the size of the data to be sent based on the previous advertised window. Therefore, if the receive buffer size is reduced before the decrease of the advertised window, the receiver may not receive a part of the data sent by the server, and the receive buffer may overflow. Therefore, *NetPIS-R* should decrease the amount of data sent from the server by

Algorithm 3 Delayed ACK Control

```
1: if a socket belongs to the background process then
2:     if the foreground process exists then
3:         size_bf ← γ * buffer
4:         ato ← (size_bf/size_obf) * ato_o
5:     end if
6: end if
```

decreasing the advertised window before curtailing the receive buffer size. The receive buffer does not decrease although the socket belongs to the background process in line 7 of Algorithm 1, which means that *NetPIS-R* waits for the decrease of the advertised window.

2) Algorithm for delayed ACK control: The advertised window control is an effective method to control network flows. However, the advertised window is just a feedback to a sender, and the final decision of the window size is decided by the sender, not by the receiver. In addition, it is hard for the receiver to know the sender's window size accurately. Thus, most of the prior work focused on changing both sender and receiver side window variables. In order to overcome the difficulty, *NetPIS* adjusts additional parameters from delayed ACK mechanism as we described in Section III.

Delayed ACK algorithm contains two major parameters. The first parameter, the bounded frame size variable, indicates the size of data that should be received prior to sending an acknowledgement packet. A receiver will not send the acknowledgement packet until it receives enough amount of data or ATO alarms. ATO is the second parameter to be adjusted for our proposed mechanism because ATO should be long enough to make a receiver collect data as large as the bounded frame size. We can decrease the data rate of the background process by enlarging the bounded frame size and ATO to certain value. The enlargement is made corresponding to the background process's $buffer$, and Algorithm 3 shows this expansion method.

In line 3 of Algorithm 3, $size_bf$ indicates the size of the bounded frame, and γ is a proportional constant we defined for the delayed ACK control and is between 0 and 1. At line 4, $size_obf$, ato, and ato_o indicate the size of the original bounded frame, the adjusted ATO, and the original ATO respectively. $size_obf$ and ato_o are constant and default values of Linux kernel. $size_bf/size_obf$ indicates the increase or the decrease ratio of the bounded frame size. Thus, a level of restriction on the background process is proportional to the background process's $buffer$. A large $buffer$ means a large bounded frame size, a large ATO, and less ACK transmissions consequently. Less ACK transmissions cause that the window size of the background process's sender increases slowly and that the wireless access point connected to the mobile device will gain more access for data transmission. Therefore, the concurrently running foreground process can benefit from the decreased number of ACK transmissions of the background processes. The effect of the delayed ACK control is evaluated thoroughly in Section V. Line 2 in Algorithm 3 is needed for *NetPIS-R* to restrict the data communications of the background processes only when the foreground and the background processes conduct data communications concurrently.

V. PERFORMANCE EVALUATION

We performed various experiments to show the practicality of *NetPIS* in terms of enhancing the performance of the ongoing foreground Internet service while maintaining the system throughput of the evaluated device that runs background services concurrently [5]. Evaluations were conducted to show the validity to control the foreground and the background processes together, the effect of delayed ACK control, and the comparison of *NetPIS* and the original kernel in terms of service quality. Constant variables mentioned in Section IV, α, β, and γ, were fixed as 0.6, 0.3, and 0.5 respectively. These variables were not optimized but proved to behave well in a number of evaluations shown in this paper.

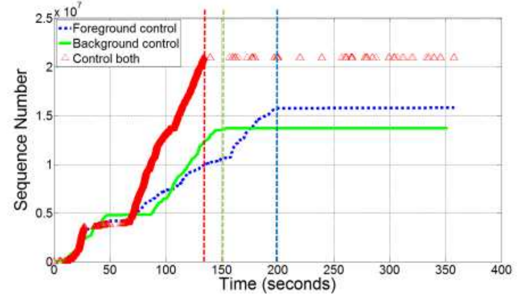
The service quality is defined in many ways in this paper. We evaluated various versions of *NetPIS* in terms of sequence number, and service delay time. The sequence number is derived by packet capture files that are recorded by a program named *shark* [17]. The service delay time is recorded by evaluating multiple devices at the same time to visualize the effect of the foreground performance isolation behavior of *NetPIS*. All devices in the same experiment are connected to the same AP to prove that *NetPIS* outperforms other devices within the same environment. In addition to the evaluation of *NetPIS* performance, we analyzed the impact of the efficient buffer control in *NetPIS*. Also, we added a simple experiment to see the low power overhead of the proposed scheme. All the evaluations were carried out on Galaxy Nexus with Android 4.3 JellyBean mobile platform.

A. Justifying simultaneous foreground and background control

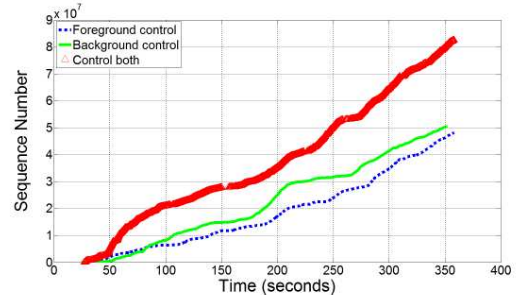
Several prior work focused on lowering receive buffer or advertised window size of low priority services only, so that the service quality of high priority service will automatically increase due to the remaining network bandwidth. However, this assumption can only be applied to the cases where there are no other devices using the wireless media. This is unlikely to happen these days where tens of public access points are installed for traffic offloading from cellular network. Immediate reaction of the foreground service is necessary to get back the network resource which is released by the background service within the same device.

In order to justify the aforementioned reasonings, we played a 2 minute long video using YouTube application as a foreground service, and performed PlayStore download of size 1.1 GB as a background service on three devices. One device controls the advertised window and the receive buffer size for both the foreground and the background service, whereas the other two devices control those parameter for either the foreground or the background service. After the video service for each device was finished, we stopped the download service of the device and examined the pcap files to analyze the increase of TCP sequence number for each service. Because TCP sequence number indicates the total size of received data, it can be a good indicator to show the network quality regardless of video service providers' own quality control techniques.

In *NetPIS*, unlike the advertised window and the receive buffer control which are used in both the foreground and the background process, delayed ACK adjustment is used to restrict only the background processes' communications. Therefore, delayed ACK adjustment is not yet applied to this evaluation to demonstrate the validity to control both the foreground and the background process together.



(a) TCP sequence number of Foreground service



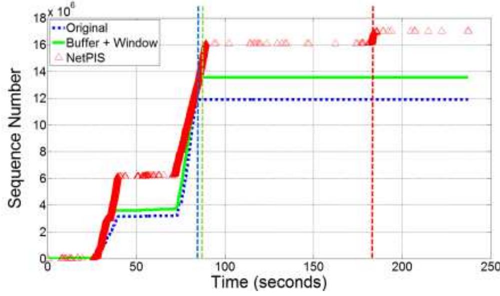
(b) TCP sequence number of Background service

Fig. 2: TCP sequence number comparison among three different types of buffer+window control (foreground only, background only, and both)

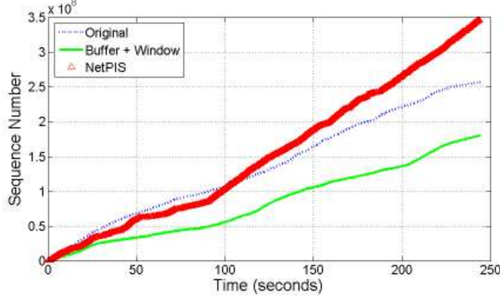
Figure 2 shows the result of the evaluation, and it is easy to see that buffer and window control to both service types outperforms the other two single service controls. The foreground-only control device shows the worst performance among the three devices since it only shifts up the buffer and window size of the foreground service without releasing any network resource that belongs to its background service. Although the device with background-only control shows better performance than the one with foreground-only control, controlling both together outperforms the others. This is because the foreground service of the background-only control device does not try to reclaim the network resource released by the background service promptly. From this experiment, we can see that both the buffer and the window size control on both services are important for quick release and recovery of the network resource within a device.

B. Analyzing the effect of delayed ACK

This part of the evaluation is shown to analyze the effect of the delayed ACK control in *NetPIS*. As mentioned before, solely modifying the advertised window size of a data receiver does not always guarantee the performance enhancement of a foreground service along with overall throughput maintenance. On the other hand, increasing bounded frame size and ATO can slow down the performance of the background service no matter how large or small the congestion window size is. At the same time, the reduced trial of background service's TCP ACK transmission will give the wireless access point more chances of releasing the downlink traffic, resulting in faster network throughput of the foreground service. In order



(a) TCP sequence number of Foreground service



(b) TCP sequence number of Background service

Fig. 3: TCP sequence number comparison among original, buffer+window only control, and *NetPIS* that includes delayed ACK adjustment

to visualize this effect clearly, we evaluated another experiment with three devices, similar to the prior evaluation. Three devices have different kernel implementation as follows: one with the original kernel, one with buffer and window control only, and one with all controls including delayed ACK.

As shown in Figure 3, the sequence number of *NetPIS* accelerates much faster than the other two devices without the delayed ACK control. In addition, *NetPIS* quickly recovers the performance of the background service after the data reception of the foreground service is almost finished at about 90 seconds. On the contrary, the background service's performance of the device with buffer and window control only is poor. Since the performance isolation with buffer and window control without delayed ACK control has the risk of killing too much traffic of the background service, the device fails to recover the performance of the background service after the end of the foreground service. Therefore, we assure that the sole control of window and buffer size is unstable, and it is necessary to apply delayed ACK adjustment to reduce such risk for the overall device performance.

Interestingly, the streaming session of *NetPIS* with delayed ACK control actually ends later than the other two. This is due to the better network status which automatically leads YouTube to switch to a better quality video that requires more data reception than the other two cases. In addition, YouTube application receives traffic maximally to collect as many video stream data as possible and rests itself until the play time almost reaches the end of the buffered video data. This is why mobile devices showed a data receiving pattern with short pauses. Although the foreground duration of *NetPIS* with delayed ACK control is longer than the other two, we can see that the total received data is much greater than the other two (about 1.4 times of the original device), which means the quality of the video is outstanding.

	Original	$NetPIS$	Service Delay
YouTube	6m 49.53s	6m 51.08s	1.55s
	1m 53.53s	1m 53.70s	0.17s
	2m 16.37s	2m 17.29s	0.92s
	2m 10.77s	2m 11.55s	0.78s
	7m 17.88s	7m 18.64s	0.76s
Daum TV Pot	1m 30.80s	1m 31.15s	0.35s
	7m 14.19s	7m 15.78s	1.59s
	1m 07.18s	1m 08.69s	1.51s
	3m 24.83s	3m 28.87s	4.04s
	2m 18.16s	2m 19.03s	0.87s

TABLE I: Performance isolation in terms of service time

C. Performance Isolation effect: in terms of service time

The evaluation of performance isolation can also be done in the perspective of service time. The service time indicates the time taken to complete the service. The service time of the original device without any concurrently running background services should be the target of *NetPIS*. To see whether *NetPIS* device with concurrently running services achieves such the target, we made an experiment configured as follows. Total 10 different videos were used as a foreground process individually, with two different video streaming services: YouTube and Daum TV Pot. Unlike TCP sequence number, the service time can be influenced by the video quality. Thus, we used videos with a fixed quality setting in this evaluation. The background service is a PlayStore download used by the prior evaluations. We evaluated the service time of the aforementioned ten videos by using two different devices: original and *NetPIS*. Original device runs the foreground service only, whereas *NetPIS* device runs both services concurrently.

The evaluation result of each video is shown in Table I, where the service delay indicates the service time difference between the two devices. If this delay is 0, *NetPIS* achieves perfect performance isolation for the foreground service. As the results show, we can see that the performance isolation of *NetPIS* is guaranteed well since most of the videos are delayed within or around 1 second regardless of the duration of the video. Although there are few cases of delay longer than 1 second, we can expect that those cases happen rarely since most Wi-Fi APs do not guarantee resource allocation fairness of different devices in some exceptional cases such as starvation, link failure, buffer bloat, and etc. Other than that, the result of the isolation technique is satisfactory.

D. Memory usage efficiency

Unlike the default setting, *NetPIS* adaptively allocates buffer space corresponding to the required buffer size of each process. In order to evaluate the memory efficiency of *NetPIS*, we tested two types of services: YouTube and web browsing. For the test of YouTube, we recorded the memory utilization from the start of YouTube application, and included additional steps to find a video such as clicking several buttons. After finding a specific video, which is 29 seconds long, we touched the play button and recorded the memory until the video was finished. About 1 minute long evaluation was made for each video play, and the same sequence was tested 5 times to

	Original	*NetPIS*	Gain
Youtube	21.75 MByte	17.13 MByte	21.24%
Web surfing	70.49 MByte	58.16 MByte	17.49%

TABLE II: Memory efficiency

	Concurrent	Foreground	Background
Original	2103.13 mW	1315.91 mW	1528.54 mW
NetPIS	2134.42 mW	1335.12 mW	1565.23 mW

TABLE III: Power overhead analysis

average the results of memory utilization. For browsing test, the default web browsing service of Galaxy Nexus was tested for 2 minutes. The 2 minute long test includes the steps of entering *m.daum.net* website, additional sequential entering of 3 links, and changing to another portal site, *m.naver.com*. This was also tested 5 times to average the results. The memory size of each case is the average memory size sampled over 1 second interval during each experiment. Table II shows the result that indicates *NetPIS* uses buffer space efficiently. It is because *NetPIS* only provides buffer space as needed, whereas default setting of Android mobile phone provides default buffer space to all the sockets without any process profiling.

E. Power overhead analysis

In order to adopt *NetPIS* to real world mobile devices, the scheme should not generate too much power waste. Using an electric power meter named Monsoon Power Monitor, we measured the power consumption of Galaxy Nexus phone with *NetPIS* implementation, and compared the result to the power consumption of the original device. Since the power of mobile device depends on the data rate, we made the environment of two devices as similar as possible and averaged the results of 5 repeated experiments for each device. The power analysis is made for three cases total: one with foreground service only, one with background service only, and one with concurrently running services. All the experiments are evaluated for 1 minute each, and the foreground service is a YouTube video stream, whereas the background service is a PlayStore download. As shown in Table III, the power consumptions of two devices are similar, and this is because the computation caused by *NetPIS* is negligible. Therefore, we can say that *NetPIS* does not affect the longevity of mobile devices significantly.

VI. Conclusions

Due to the development of mobile devices and communication technology, not only the foreground service but also the background services often generate data communications at the same time in a mobile device. Therefore, the network performance of the foreground service frequently suffers from concurrent data communications of the background services. In order to resolve this problem, this paper proposed *NetPIS* which provides network performance isolation for the foreground service by receiver based control. A mobile device using *NetPIS* can manage downlink wireless traffic in terms of processes and guarantee the network performance of the foreground process when multiple Internet services run concurrently. Unlike the existing performance isolation methods executed by virtual machines or flow based mechanisms with limited scalability, *NetPIS* uses network trace information and adjusts receive buffer size, advertised window size, and delayed ACK parameters in order to provide network performance isolation with low overhead. The performance of *NetPIS* was evaluated by in-depth analysis with various experiments using a real mobile device. Indeed, *NetPIS* is proved to behave well with frequently used services such as video streaming and Internet browsing services [5].

Acknowledgment

This research was supported in part by Digital Media & Communications R&D Center, Samsung Electronics, and in part by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (NRF-2013R1A1A2010388).

References

[1] Ericsson, "Ericsson mobility report, on the pulse of the networked society," June 2014.

[2] "NetPIS - Posing problem," http://youtu.be/O9vjuIeOw4M/, 2015, [Online; accessed 07-September-2015].

[3] A. Shieh, S. Kandula, A. Greenberg, and C. Kim, "Seawall: performance isolation for cloud datacenter networks," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*. USENIX Association, 2010, pp. 1–1.

[4] Y.-K. Lim, C.-G. Kim, M.-S. Lee, and S.-D. Kim, "The stack allocation technique on android os," in *IT Convergence and Security 2012*. Springer, 2013, pp. 727–732.

[5] "NetPIS - Performance Evaluation," http://youtu.be/qiFk7n8eCnA/, 2015, [Online; accessed 07-September-2015].

[6] S. Chen, Z. Yuan, and G.-M. Muntean, "An energy-aware multipath-tcp-based content delivery scheme in heterogeneous wireless networks," in *Wireless Communications and Networking Conference (WCNC), 2013 IEEE*. IEEE, 2013, pp. 1291–1296.

[7] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel, "The cost of a cloud: research problems in data center networks," *ACM SIGCOMM computer communication review*, vol. 39, no. 1, pp. 68–73, 2008.

[8] A. Khan, A. Zugenmaier, D. Jurca, and W. Kellerer, "Network virtualization: a hypervisor for the internet?" *Communications Magazine, IEEE*, vol. 50, no. 1, pp. 136–143, 2012.

[9] Y. Dong, X. Yang, J. Li, G. Liao, K. Tian, and H. Guan, "High performance network virtualization with sr-iov," *Journal of Parallel and Distributed Computing*, vol. 72, no. 11, pp. 1471–1480, 2012.

[10] Y. Etsion, D. Tsafrir, and D. G. Feitelson, "Process prioritization using output production: scheduling for multimedia," *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, vol. 2, no. 4, pp. 318–342, 2006.

[11] H. Zheng and J. Nieh, "Rsio: automatic user interaction detection and scheduling," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 38, no. 1. ACM, 2010, pp. 263–274.

[12] J. Semke, J. Mahdavi, and M. Mathis, "Automatic tcp buffer tuning," *ACM SIGCOMM Computer Communication Review*, vol. 28, no. 4, pp. 315–323, 1998.

[13] T. Kelly, "Scalable tcp: Improving performance in highspeed wide area networks," *ACM SIGCOMM Computer Communication Review*, vol. 33, no. 2, pp. 83–91, 2003.

[14] A. Venkataramani, R. Kokku, and M. Dahlin, "Tcp nice: A mechanism for background transfers," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 329–343, 2002.

[15] Y.-C. Chen, D. Towsley, E. M. Nahum, R. J. Gibbens, and Y.-s. Lim, "Characterizing 4g and 3g networks: Supporting mobility with multipath tcp," *School of Computer Science, University of Massachusetts Amherst, Tech. Rep*, vol. 22, 2012.

[16] R. Braden, "Rfc-1122: Requirements for internet hosts," *Request for Comments*, pp. 356–363, 1989.

[17] K. Meng, Y. Xiao, and S. V. Vrbsky, "Building a wireless capturing tool for wifi," *Security and Communication Networks*, vol. 2, no. 6, pp. 654–668, 2009.