CS530 Critical Review

Preparing Next Mobile Explosion: a *glimpse* of multi-device operating system

Tae Gyeong Lee *20130505*, Youngjae Chang *201305551*
{danny003, youngjae.chang}@kaist.ac.kr

1. Introduction

Mobile devices (e.g. Smartphone and tablet PC) have crossed a tipping point. Since 2014, there are more mobile users than desktop users. About 3 billion people are using smartphone, accessing internet. Mobile devices had been adopted explosively. It tooks 2.5 year to reach 40% penetration from initial 10% traction. Internet and Radio took about 10 years to do the same thing.[1]

A trigger that ignited the explosion was mobility. Not bounded by time and place, mobile devices became essential parts of human life as the most effective and convenient communication tools. The ubiquitousness allow apps to provide richer user experiences. The apps more or less communicate with servers via wireless connection and sense ambient environment using on-device sensors. Average mobile user uses 80 applications on their smartphone. App economy projected to generate revenue of $50.9 billion this year (2016).[2]

What's next? Current mobile devices are governed by two infamous operating systems: iOS made by Apple and Android made by Google. They are the first and second biggest companies by market capitalization in the world (2016), respectively.[3] Drawing a blueprint of next mobile explosion is thus important. Having right idea and executing it in right way will leads us to the next most valuable company in the world, shaping next technological landscape.

In this paper, we focus on the increase of the number of devices per user as a probable source of the next mobile explosion. More and more digital apparatus are introduced with keyword 'Internet of Things', having an ability to connect on wireless network. User expects to have a seamless experience over these devices. The multi-device environment breaks few assumptions existing on previous approach and proposes new challenges on designing operating system.

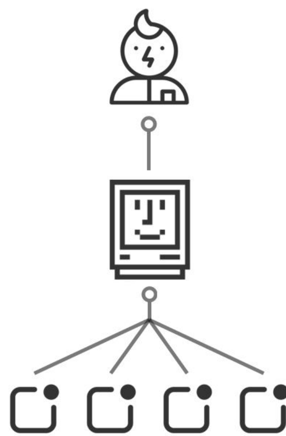
---

[1] https://www.technologyreview.com/s/427787/are-smart-phones-spreading-faster-than-any-technology-in-human-history/
[2] https://www.appannie.com/insights/market-data/app-annie-releases-inaugural-mobile-app-forecast/
[3] https://en.wikipedia.org/wiki/List_of_public_corporations_by_market_capitalization#2016

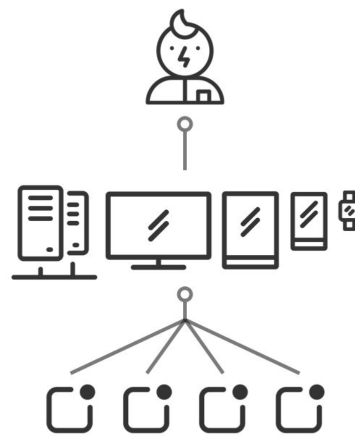
The review presents a crude glimpse of a future mobile operating system, based on 3 papers. Section 2 briefly reviews current approach towards mobile systems, and points out assumptions that may fail as user needs matures. In Section 3-5, we conduct case studies on 3 papers which actually succeeded to compensate the new user needs extending current operating system. Section 6 discusses about future research directions. Finally, we summarize and conclude the review on Section 7.

2. Multi-Device Operating System

Large deployment of mobile devices have introduced new challenges on designing mobile systems. These challenges are resulted from broken assumptions of previous designs: User interacts with one device at a time. In this section we highlight 6 of them.



(a) one device - multiple apps



(b) multiple devices - multiple apps

2.1. Challenges

Multi-Surface User Interaction

Multi-Surface The most fundamental assumption that we try to tackle is that user uses single device at a time. Nowadays user interacts with multiple devices at a time. We jump to smartphone to communicate with others, and picks up the notebook to work on the remote place. When jump from one device to the other, we often face completely different environment, not maintaining previous context. However, modern users expect continuous experience across multiple devices. The success of cloud storage service (i.e. Dropbox) can be considered as an early signal of these needs.

App State Migration To support multi-device environment we should make application state to flow among devices. Even though more and more applications are supporting synchronized settings over the devices, the continuity feature was not a service provided by operating system. In operating systems' view, program state should stay inside a device, inside a memory. Contrary to the design, more and more apps require personal settings and files to be synchronized. Moreover, user wants application *state* to be

shared, i.e. pick up the smartphone and start working on the exact point I left on laptop. To implement application state migration in operating system level, we should make application data and program state flow among devices seamlessly.

Device Heterogeneity

Sharing Computing Resources Moreover, an application or a portion of application are being executed in heterogeneous devices. A mobile device should be able to carry around. This imposes severe limitation on its weight and size. The physical constraints result in limited battery and computation performance. To overcome this obstacle, people are trying to utilize nearby computational resources to accelerate application. This can be harder than it sounds, since the application should be run on top of multiple form factor, we have to deal with device heterogeneity — different ISAs, hardware configuration and system states.

Coping Variety in Capability Also mobile devices have larger variety in their capability. The mobile devices can be divided into two major category: smart multi-purpose devices and dumb single-purpose devices. Not all mobile devices will have expansive computational resources. To serve a multiple application on dumb devices, a system would be able to be run on even dumb devices, abstracting lack of capability in certain ways. The variety can also exists on screen size. The application should be able to be customized according to screen size, while sharing its business logic.

Userful Service

Loosely-coupled Hardware Considering newly introduced IoT devices, notion of tightly linked physical device have to be also tackled. The computer is actually a composition of multiple hardware components communicating on predefined interface. These devices are usually linked to mainboard directly, giving fast and lossless connection. However, modern hardware are not packed into a box. We may access sensors embedded in home environment (e.g. air quality sensors) via wireless communication as if it's plugged into a personal device. Having loosely coupled structure, we should be more aware about a communication failure scenario.

Privacy in Context Finally, we should interpret privacy more carefully. Until multi-device environment, a privacy is more or less synonym of the protection. Secure system blocks all the unapproved access on the system. However, in upcoming mobile environment, the privacy would be more about making user data *flow* as intended, preventing a leakage or an interception. If we focus only on protection, the resulting design will innately restrict sharing between devices into a few predefined patterns. To enforce privacy with finer granularity, we may need to semantically understand the user data flowing in and out of devices communication channel.

2.2. Motivating Scenario

Larry is lying on his bed, playing Mario Kart on his phone. Jacob, Larry's younger brother, knocked on the door. Larry ask to play with him. They went to living room. Larry migrate app to TV, to use a bigger screen. Larry and Jacob pairs their smartphone with TV, setting it as a controller. Since Mario Kart is a 3D game, TV find that it does not have enough computing resource to render 3D graphic with such a high resolution. TV utilizes idling nearby desktop to accelerate the rendering process. Larry and Jacob enjoy fluent game play, without noticing complicated communication happening in between devices.

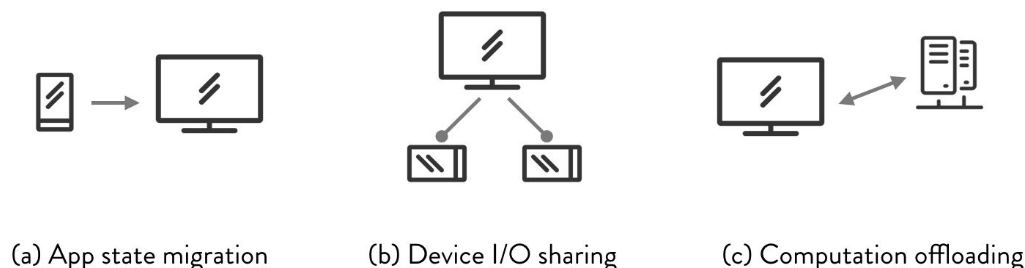
The above scenario shows how newly developed technologies can be integrated to build up a seamless user experience. First, we can spot an app migration process. App state is migrated from Larry's phone to TV. We have to remember that app running on TV is being executed with Larry's context. For example, if larry bought a new map using in-app purchase, the map should be able to played within TV application. When the app is terminated, all the sensitive information should be deleted from TV.

The scenario also demonstrates pairing of multiple devices. Smartphone's sensors — touch panel and gyroscope — are used as a controller for Mario Kart. The sensor information are transmitted using wireless network. Having appropriate abstraction for these I/O devices make the process more fluent. TV may need to contact speakers, CCTV cameras, and other home appliances. Leaving this scenario to app developers' burden will eventually fragmentize the user experience.

Lastly, we can observe application acceleration using idling computing resources. Mobile devices often have hard constraints on its physical size, having limited computing power. Future mobile system should able to cope this problem by abstracting nearby computing resources as a form of small computing cloud. By offloading a computation to more powerful devices, we can earn benefits on battery usage and UI latency.

2.3. Problems and Attempts

Previously, we've discussed about challenges resulted from an increase of the number of devices per user. To support multi-surface environment, we should be able to convey application states across devices. Moreover, since devices have different specifications, we should properly abstract these variety. Finally, we should loosen the boundary of device while enforcing security on the data flow.



(a) App state migration (b) Device I/O sharing (c) Computation offloading

These challenges had been realized using an real-life scenario. We've pointed out three technologies: (1) app state migration, (2) device I/O sharing and (3) computation offloading. The following sections will show existing attempt on each problem.

Flux tackles a problem on app state migration, which related to multi-surface user interaction design. Flux solves problem by completely migrating application to paired device, including source code and system states. Rio, which will be discussed in section 4, suggests I/O sharing between devices, loosening the boundary of a single device. By replacing device file with virtual device file, user can send sensor data over the wireless network. Finally, Flip-Flop Replication challenges computing resource sharing problem with novel viewpoint of replication. By replicating system on the server, and changing leaders between two replicas, user enjoys best of two cases: better latency on user interaction phase and better speed on computing or networking phase.

3. Flux

Due to proliferation of users owning multiple devices with various shapes, sizes, and purposes, the demand for multi-surface applications that run seamlessly across multiple user devices are expected to increase. Pre-existing general system solutions are focused on screencasting or cloud-based approach, but screencasting cannot utilize additional resources or capabilities of new devices and cloud-based approach greatly suffers from network connectivity and privacy issues. As an alternative, “Flux: Multi-Surface Computing in Android” [4] suggested an app migration approach that enables apps to become multi-surface.

Since Flux handles app migration across multiple devices, device heterogeneity is the main challenge of the Flux system. Screen sizes, platforms, OS versions, hardware sensors, wireless technologies, and possible device states vary in manufacturer, years, etc. Such device heterogeneity leads to residual dependency challenges in two ways: (1) app’s interactions with shared system services and (2) device specific and app specific environments. In Android systems, each application’s activities and services interact with system services to access hardware components with Android framework, and it is not feasible to offload whole system service to guest device. The second challenge is that apps might contain device dependent state which is not accessible to some devices. To address these two main challenges, Flux used two techniques: Selective Record/Adaptive Replay and Checkpoint/Restore In Android.

- **Selective Record/Adaptive Replay:** Since the migration of system services interacting apps is not feasible, Flux leads guest device’s system services to take over after app migration. Checkpoint/restore approach could be one solution, but making checkpoints and restoring all systems services’ state is heavy work. Instead, Flux records all calls updating system services’ app-specific state in the home device and deterministically replay in the guest device with its own system services. The system services support a wide range features while mobile devices’ resource is limited and some services might not be available or not identical across devices; therefore, Flux selectively records method calls of system services and adaptively replays it. To support selective record and adaptive replay, Flux modified common system services with decorators indicating which methods are recorded or dropped in specific condition.

- **Checkpoint/Restore In Android:** CRIA checkpoints critical user-level and OS-level state, and restore it on guest device. It deals with device specific state and core app state (app-specific state) in Android driver. Flux leverages Android app initialization mechanism in restore step so that device states is reconstructed in a manner customized to the guest platform.

The evaluation of the work tested for different devices, mobile device and tablet pc, and Android versions with popular apps. They focused on the amount of power, time, and data overhead during app migration. Since Flux provides recorded logs and checkpoints, the amount of data transferred is not negligible; therefore, it spends most of time during migration in data transfer. Moreover, Flux should transfer the whole apk file of app in pairing step before migration. In contrast, the power and CPU usage overhead were negligible. The work also analyzed apk file size of Google Play apps to emphasize that data transfer overhead is not a serious problem, and additional optimization of Flux and setting better network condition would improve user experience.

---

[4] Alexander Van’t Hof, Hani Jamjoom, Jason Nieh, Dan Williams, “Flux: Multi-Surface Computing in Android”, (EuroSys ‘15)

Flux is a sound system that migrates apps across heterogeneous Android devices without requiring any modification of the application. Yet, the prototype implemented in their work has low performance and there are some limitations of Flux system. Since Flux records or checkpoints app states and IPC calls, apps requiring OpenGL context to persist in the background, multi-process, or common SD access are not supported by Flux. During interaction with content providers of Android framework, Apps cannot be migrated through Flux. As their evaluation has shown, app migration latency of Flux is long. The demand of migration delay might vary in situations, but more than 10 second delay will lead to bad user experience. In addition, Flux focuses on only app migration for multi-surface applications; therefore, utilizing multiple devices at the same time is out of scope. When app is migrated, Flux enforces fome device to run app in background and continues the execution on only guest device. The computation power and other hardware components of home device are not accessible which infers that home device is wasted. Since utilizing multiple devices for richer environment and better resources is one of main functions of cloud computing, utilizing home devices might be additional research direction for Flux.

4. Rio

Nowadays, an user owns a variety of mobile systems, each equipped with a diver I/O devices, such as cameras, speakers, microphones, sensors, and cellular modems. Having different forms, mobile devices have their own distinct sets of I/O devices. Allowing users to remotely access these I/O devices enables many inspiring scenarios: We may take picture on smartphone using remote tablet's camera, we may play racing game on tablet using phone's tilt sensor or we may serve mobile devices a wireless connection using one sim card.

Existing solutions, i.e. IP Webcam which shares camera between devices and MightyText that shares keyboard input, are implemented on application layer. This result in limitations: remote I/O implemented by these application cannot support unmodified applications. Also it do not expose all functionalities of an I/O device for sharing. Only a few selection hand-picked by application developer will be served. We should also note that these applications are constrained to those class of I/O devices, MightyText cannot be used to share camera view.

"Rio: A System Solution for Sharing I/O between Mobile Systems"[5] solves the problem by abstracting remote I/O devices as device files. Device files are abstraction used by unix system to represent I/O devices. User program can conduct normal file operations on the device file, then the kernel converts it to I/O operation using device driver. Rio succeeded to abstract remote I/O beautifully, honoring legacy structure deeply integrated into operating system.

Rio picks up three major challenges arose upon abstracting remote I/O into a virtual device file. First, a device driver reside in two different devices — in two different memory spaces. We should make sure all the local I/O operation have same effect on virtual device driver, shadowed in remote system. Secondly, Rio have to reduce an impact of RTT between two devices. To make a synchronous I/O operation, typical operation is done with 4 steps: I/O control signal, copying from user, copying to user and another I/O control signal. In remote I/O scenario, those 4 steps result in 6 synchronous network requests. Generating large overhead if RTT is big. Finally, since remote I/O devices are linked via network connection, we should safely handle unreliable network connections. To address these challenges Rio proposed two major techniques.

---

[5] Amiri Sani, Ardalan, et al. "Rio: a system solution for sharing i/o between mobile systems.", (MobiSys '14)

- **Distributed Shared Memory(DSM) Synchronization:** Since memory copy is a large burden for CPU, I/O devices write their sensor data via direct memory access(DMA) technique. User process read the data without copying using mmap command. To support this operations, Rio maintains shadow page on client where user process is running. But since memory copying is very expensive, Rio uses write-invalidate strategy. Compared to proactive update strategy which transfer data whenever new data is available, Rio transfers data only when remote read occurs or the remote copy is outdated. To do this, Rio catches all the memory operations conducted on shadow page region, and convert them to appropriate network request.

- **Pre-copying and batching:** OS assumes the latency between I/O device and CPU is very low so usually I/O operation is done synchronously with multiple round trips. To reduce impact of RTT due this multiple roundtrips. Rio aggregates network communications via pre-copying and batching. Since copying from user does not need I/O's up-to-date information, Rio pre-copy the operation result to server device. Similarly, since copying to user is about remote I/O data, we batch the result into client device. Thus we can reduce the impact of RTT.

- **Policy for disconnection:** Rio does not propose novel approach on this challenge. Server device which provides I/O data kills proxy process as if killing local process. On client side, it first try to switch to local I/O devices if possible. Not much word on latency, since latency does not affect correctness of operation, at least.

Though Rio provides an elegant abstraction of remote I/O devices, Rio still have its limitations. First, the implementation of DSM scheme and Kernel device driver is very complicated. Though application designer can enjoy the abstractions, applying the research directly to market seems unlikely especially for hardware developers. Second, kernel layer abstraction doesn't fit on Android. Since android application have no direct access to device files, these operation are usually proxied via application layer daemon process. In case of Android, implementing remote I/O on this upper layer — Hardware Abstraction Layer (HAL) — may be a simpler way to achieve same concept. Since device files does not have security feature other than file permission, remote communication is prone to attacks. Finally, the most severe limitation is its performance. Rio failed to reliably stream 128x96@30fps video. Author accuses network bandwidth for the poor quality, but there are still many untried techniques which will leads to better quality while using same bandwidth — i.e. compression. Solving these issues will clean up the roads towards more efficient I/O sharing system.

5. Flip-Flop Replication

Mobile devices have constraints related to size concerning its mobility; therefore, mobile devices have greater battery limitation, less computational power, and poorer network condition than other computing devices. Therefore, computation offloading, in which heavy computation of mobile application migrates to resourceful servers and thus receives results from servers, is one of the main topics in mobile systems.

The main goal of computation offloading is to reduce burden of mobile device's computation to improve performance and to save energy. Most existing computation offloading techniques use partition based approach which divides the program into two parts: mobile execution part and offloaded part. The partition based offloading has some critical challenges. Since it runs a part of application in the server with various architectures and processors, the system should support heterogeneous environments and machines, and annotate which portion should be offloaded. Also, the offloading system should be aware of

the context of application and network condition for optimization. Annotation and context awareness are difficult problems, and developers are responsible for these problems to solve. Wireless condition is not stable due to mobility of devices, so fault tolerance is one of the main issues in computation offloading, and some privacy issues also occur since the app components are floating in the air.

To address these problems, “Accelerating Mobile Applications through Flip-Flop Replication” [6] introduces Tango, a flip-flop replication system which replicates application and executes on both mobile device and the server. Replication approach simply remove the needs of annotating partition and raising awareness of device’s context because it replicates whole app code without partition and always executes replicas on both client and server sides. Since apps with Tango are executed in both sides, Tango should ensure that both replicas will produce the same results. Therefore, non-determinism of execution became a new challenge for replication system. For non-determinism challenge and remaining challenges of partition based computation offloading, Tango used replicated/non-replicated portion architecture, deterministic replay, leader switching, and server fail recovery techniques.

- **Architectural solution:** Tango consist of two parts: replicated portion and non-replicated portion. Deterministic execution is able to run on server without any connection with mobile client and make same results on both replica. Therefore, deterministic components are included in replicated portion: Dalvik VM, storage subsystem, native method in standard libraries, and UI stack. Tango system replicates whole Dalvik VM which is device agnostic to achieve inter-operability. The non-replicated portion includes external determinism of app execution: IPCs, I/O performing methods, and native methods, called only once in mobile client side or in server side. Each native methods such as device sensor I/O, user I/O, or network communication are pinned in client or in server and executed in native threads. The internal non-determinism such as random methods or asynchronous scheduling also included in non-replicated portion. Through these architecture, Tango removed external and internal nondeterminism.

- **Deterministic Replay:** Tango use deterministic replay technique (leader decide and recode / follower replays) for execution of the non-replicated portion. The native methods in non-replicated portion are executed by replica that reaches first and give same results and states to replica that follows after leading replica. Internal nondeterministic parts are decided by leading replica and following replica also replays according to the log created by leading replica.

- **Leader Switching:** Since Tango uses deterministic replay, deciding which replica to be leader, which is responsible to run native method and decide internal non-deterministic execution. Leader switching costs only a single RTT; therefore, it makes sense that the switching should be done according to appropriate app phase. Tango heuristically monitors frequent user I/O, network communication, and native method calls to decide appropriate leader, and Tango also predicts heavy computations. Also, to make switching overhead small, Tango rejects leader switching while follower is lagging far behind from the leader. Tango uses heuristic ways yet for leader switching, and it is their future work to make fine automatic leader switching algorithm.

- **Server Fail Recovery:** Computation offloading communicates various app states and results during app execution; therefore, it suffers from server failures and network connection breaks. In case of Tango, if server fails when the server replica is leader, follower replica in mobile client

---

[6] Mark S. Gordon, David Ke Hong, Peter M. Chen, Jason Flinn, Scott Mahlke, Zhuoqing Morley Mao, “Accelerating Mobile Applications through Flip-Flop Replication”, (MobiSys ‘15)

cannot follow leader's decisions. Therefore, Tango simply manage log-based backup server and fet non-deterministic execution results and restart server side when server fails.

They evaluated Tango for computation or network intensive applications to measure the performance gain and energy effect by using Tango. According to their results, Tango can lead about 2-3x performance improvements and small amount of energy save. They also emphasize that Tango is beneficial for network intensive applications since the server has much better network connection than mobile device. In case of partition-based computation offloading, frequent communication between client and server side due to app states transfer made the system weak for network intensive applications when RTT value is big. Besides, Tango's both replicas communicate in pipeline style and the server replica will be leader for network intensive application; therefore, Tango also improves performance although RTT value is big.

Since deterministic replay technique, Tango is able to improve performance without predictions or profiling of device's context, and it is one of strong points of Tango. In addition, Tango replicates whole Dalvik VM to make replicas; therefore, no modification or labeling of previous applications are needed. Since applications are executed in both replicas, data produced during app execution and various app states do not transfer between client and server. This makes low data transfer overhead, and it is beneficial for privacy issues since less data is transfer in the air.

The most work about Tango is removing nondeterminism, but techniques used to remove nondeterminism make limitations of Tango. Applications with multi cores and multi process cannot be supported by Tango. Since non-deterministic portions are not replicated, applications with heavy external non-determinism (frequent native method call) are hard to be supported by Tango. As the author of the paper mentioned, immature leader switching algorithm acts as limitation. In addition, privacy and security problems are still not solved yet. This issue might be one of required future topics for computation offloading systems.

6. Possible Research Directions & Open Issues

Multi-device environment became a new trend due to increase of various device usage, and new assumptions and demands will appear as mentioned in Section 2. Through case studies described in Section 3-5, we figured out main challenges and techniques related to app migration, I/O sharing, and computation offloading across devices. These works contribute to development of multi-device system, but have some unsolved limitations. Also there are still some issues which should be addressed. In this Section, we discuss additional topics and some research directions that will be issued in future.

6.1. Privacy

Multi-device operating system tights multiple devices by network connections. Therefore, method calls, produced data, app state, or sensors values can be transferred by wireless communication, and thus private data leakage may occurs. Flux, Rio, and Tango, described in Section 3-5, did not addressed privacy challenges, and it still remained as a limitation of these works. The three works in case studies achieved their own goal without any app modification, which infers that the app developers do not have to consider multi device environment. Therefore, some app developers might not able to realize how data is transferred in network communication or how app permission affects in other devices. Consequently, developing new permission scheme and encryption considering multi device should be addressed in future works.

Flux, Rio, and Tango focused on utilizing multi-device of a single user, while multi-device environment might not be limited for a single user. Multiple users can share I/O of their devices such as scenario described in Section 2 or a user can borrow computation power of other users' device or public cloudlet. To prevent malicious devices to be integrated in multi-device system, pairing policies and interfaces should be designed.

6.2. Seamless User Experience

Using multiple devices provide rich resources and environment so that users can utilize devices flexible according to various purpose and enjoy useful services. To provide better user experience, it is important to make illusion that a user is interacting with interface of whole multi-device system and not aware of complex architecture and network connection of the system. Therefore, time overhead directly affects user experience, and providing seamless user experience is remained as future works. Especially, app migration, issued in Flux, should have negligible migration latency since user will feel interruption or stop during app use if time overhead is too big. In Flux, Rio, and Tango, the system make a pair or a group before operating their functionalities. Although they did not show evaluation of overhead related to pairing or grouping, it seems to spend quite long time. These presetting state latency will also ruin user experiences.

6.3. Wireless Network Congestion

Unlike wired network connection, wireless technologies share same media, the air, and they are difficult to controlled since wireless signals propagate broadly. Therefore, allocation of bandwidth of wireless network is much difficult work than wired network. In case of multi-device environment, the number of devices sharing same air is big, and it will be greater in public places such as cafe, conferences, or universities. Moreover, long latency due to network congestion will lead multi-device system to be useless since it tights devices via network connection.

There has been many works done about relaxing allocation competition and cross technology interference techniques such as relay nodes, beamforming by MIMO, and modification of MAC protocols to make better network throughput. Using these techniques can improve multi-device system's performance. Also optimization of data transfer might be next research topic addressing these issues.

7. Conclusion

Nowadays, an user owns multiple devices including mobile phone, tablet PCs and other computing devices. Greeting the trend, some works have done to provide richer user experience and computation resources. The works include app migration, sharing hardware I/O, computation offloading. We considered these multi-device system as a probable source of next mobile explosion. Flux, Rio, and Flip-flop replication are reviewed as a representative examples of such system. Flux demonstrated seamless user experience among multiple devices, Rio loosened boundary of device allowing it to connected via wireless network and Flip-flop acceleration showed acceleration of application using outer resources without modifying a single line of code. We also pointed out common challenges of these systems and outlined future research directions. Firstly, privacy and security issues related to data leakage and malicious devices existed on all three cases. Next, reducing overhead have been selected as a key challenge for a seamless user experience. Lastly, we emphasized wireless network congestion as a considerable topic that should be researched as devices are tightly integrated via wireless connection.