RE-NetBoot: Resource-Efficient Network Boot for IoT Platform

Keon-Ho Park and Ki-Woong Park*
Department of Information Security, Sejong University
Seoul, Republic of Korea
imguno0629@naver.com, woongbak@sejong.ac.kr

Abstract

The Internet of Things (IoT) platform consists of numerous IoT devices and a small number of servers. In order for the platform to operate efficiently and reliably, servers need to have complete control over many IoT devices. The advantage of using network boot in this environment is that the server has control over the device's boot process. When applying network boot on an IoT platform, the server needs to be able to handle the network boot that many IoT devices request. If the server does not have enough processing power for the network boot request, IoT device boot would be delayed. In this paper, we propose a resource-efficient network boot to solve this problem. The proposed framework has three mechanism. First, the server reserves the right to control which system image to transfer during the network boot process on the platform. Second, the server applies deduplication to the system image to minimize the amount of data required for transmission. Third, the server monitors network boot requests and performs file transfers appropriate to the situation, minimizing time spent on transfers. The proposed system allows the server to efficiently control numerous IoT devices when using network boot on the IoT platform.

Keywords: Network Booting, Resource-Efficient, IoT Platform, Data Deduplication, Scheduling

1 Introduction

According to Gartner, more than 20 billion devices will be connected to the Internet by 2020 [5]. Already an infrastructure consisting of many IoT devices, such as smart homes, smart buildings, and smart cities is deployed worldwide. IoT devices feature low cost and low power, so IoT devices have lower specifications than regular systems. Despite these limitations, we want to apply a network boot to get the most out of the efficiency of IoT devices. Network booting uses a network file server to boot a system instead of the local disk drive. Network booting can reduce production costs by reducing the local disk drive capacity required for each device. In addition, building such an infrastructure facilitates the deployment of different operating systems across the network. Every time a device is booted, the operating system can be easily changed to reduce the operating costs by allowing one device to be used for multiple purposes. Because of these advantages, network booting is widely employed in educational environments. We want to apply the benefits of network booting to the IoT infrastructure to flexibly change the operating system of IoT devices to leverage their efficiency fully. In addition, the flexible and versatile use of each device can reduce infrastructure operating costs and improve the utility of the infrastructure.

There are some issues involved in applying network booting to the IoT infrastructure. First, network booting is driven by the device. The device explicitly requests the server for the required system image. This mechanism makes it difficult for the server to control the booting of the device. The passiveness of

*Corresponding author: Dept. of Computer and Information Security, Sejong University, Neungdonr-Ro 209, Gwangjin-Gu, Seoul, Republic of Korea, Tel: +82-2-6935-2453

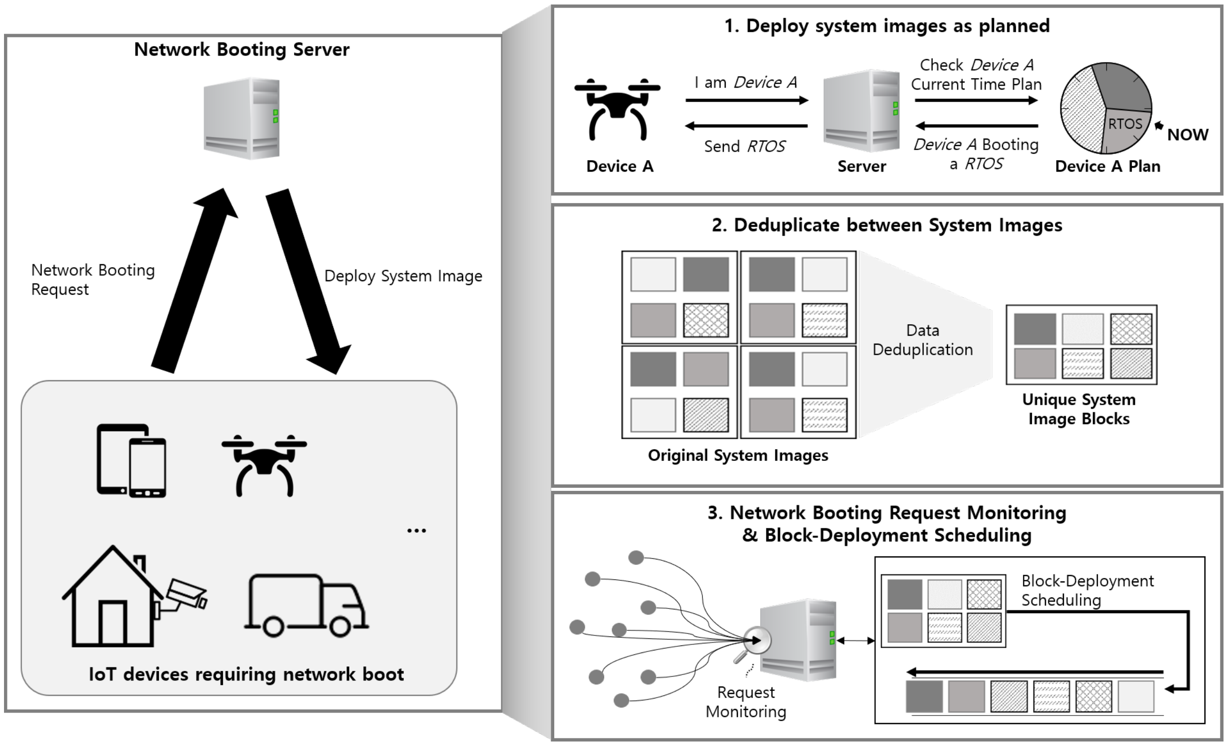

Network Booting Server
Network Booting Request
Deploy System Image
IoT devices requiring network boot
1. Deploy system images as planned
I am Device A
Send RTOS
Check Device A Current Time Plan
Device A Booting a RTOS
RTOS
NOW
Device A
Server
Device A Plan
2. Deduplicate between System Images
Data Deduplication
Original System Images
Unique System Image Blocks
3. Network Booting Request Monitoring & Block-Deployment Scheduling
Request Monitoring
Block-Deployment Scheduling

Figure 1: Three Main Mechanisms of RE-NETBOOT

the server increases the difficulty of dynamically control system images as more IoT devices are operated. Second, an infrastructure with network booting will have more network traffic than one without it. The process of deploying a system image via the network increases traffic across the entire infrastructure. Increased network traffic may make the system more prone to the disadvantages of network booting. As the number of IoT devices in operation increases, the network traffic required to supply system images will surge, possibly exceeding the available bandwidth. Finally, in an infrastructure with many IoT devices, network booting requests may occur more frequently than in smaller infrastructures. This increases the proportion of server resources that are consumed by handling network booting requests. As the number of IoT devices in operation increases, a centralized control method for controlling the system images will hinder scalability. In addition, delays in response to network boot requests can cause delays in the operation of the infrastructure, which can lead to infrastructure paralysis.

To solve such concerns, we propose RE-NETBOOT, resource-efficient network boot. RE-NETBOOT consists of three main mechanisms as shown in Figure 1. First, the device does not explicitly request a system image. The server deploys a system image in response to network boot requests from devices based on an infrastructure operations plan. The device sends its device serial number (SN) to the server. The server verifies the identity of the device and sends a system image to boot the device according to the operations plan. This means that the server is centrally controlled for all the devices. Second, the file distribution is enhanced by deduplicated system images that remove redundancies in the system image blocks that comprise the system image, minimizing network traffic during system image deployment. The server sends a deduplicated system image block in response to the network boot request, and the device recovers the original system image. Our insight is that there is redundancy in data generated on the same base. Many operating systems used for IoT devices are based on Linux, which contains such redundancies. To accomplish deduplication, we designed an indexed file format and file transfer protocol for file recovery. we design the Index File format to recover a deduplicated system image. Third, the server monitors network boot requests in real-time and dynamically changes the way files are distributed, minimizing the time to complete system image deployment for requests. By default, file distribution is handled as a unicast transmission. However, when network boot requests are concentrated to a certain degree, file distribution is handled as a broadcast transmission. We also determine the order in which the system image blocks are sent when broadcasting them, minimizing the average deployment completion time, which in turn minimizes the time it takes to deploy the system image in response to network boot requests. The contributions of this work are as follows: First, even with more IoT devices in operation, the server can dynamically change the system image for every device it controls. Second, we combine deduplication with network booting to minimize network traffic for system image deployment. Third, the proposed framework can minimize the response time dynamically according to the network boot request situation.

The remainder of this paper is as follows. Section 2 introduces related works, and Section 3 describes the proposed framework. In Section 4, we describe a testbed construction for future experiments, and conclusions are made in Section 5.

2 Related Work

2.1 Network Booting

Network booting boots a remote system using a network file server instead of a local disk drive and is typically used in diskless systems without local storage. These systems use the preboot execution environment (PXE) to receive an internet protocol (IP) address from a dynamic host configuration (DHCP) server and then receive the files to boot the system. The typical protocol used for file transfer at network booting is TFTP. TFTP is designed with minimal functionality to perform file transfers with very little memory [12]. In addition to TFTP, iPXE, an enhanced, open standard for network booting, also supports HTTPS-based file distribution [7]. In this paper, we propose a design based on TFTP that minimizes the memory footprint to support network booting for the IoT, which is comprised of many low power and low specification devices.

TFTP was first proposed as a unicast transmission method. To address the bottleneck that occurs when multiple devices request files from a single image server, the multicast option [2] was added, and a format that supports broadcast transmissions [1] has also been proposed. In addition, previous works have proposed a file distribution method using the peer-to-peer method [11, 13], which provides a solution to bottlenecks by handling file transfers for a single type of file. However, it does not cover various types of boot file transfers. In this paper, we designed RE-NETBOOT to overcome the bottleneck that can occur when various devices request different kinds of boot files.

2.2 Data Deduplication

Data Deduplication is a technique that compresses data by eliminating redundant areas of a dataset. Deduplication reduces the data footprint by removing data duplication in or between files. Deduplication is primarily used for backup storage [9] because it backs up only the parts that have changed stores the parts that have not changed. This approach extends the capacity of the network by reducing the network traffic incurred when file contents between distributed file system clients and servers are synchronized [4, 10] .

There are two classes of data deduplication that operate at different data levels [3]: block-level deduplication and file-level deduplication. This white paper focuses on block-level deduplication to eliminate redundancy between multiple operating system files stored on network file servers.

The data deduplication process is as follows: First, deduplication algorithms divide the data into chunks. The chunk sizes used to divide the data include fixed- and variable-length chunks [8]. The fixed-length chunk method divides a file into fixed-length blocks, while variable-length chunks are different sizes that divide the file into blocks. Deduplication then computes a hash value for each chunk and compares these hashes to find matching chunks or compare chunk values directly. We used a fixed-length chunk approach in which the chunk length was set equal to the size of the data block used by the file transfer protocol for compatibility.

3 Design of Resource-Efficient Network Boot for IoT Platform

The proposed framework has four phases and one file deployment protocol, as shown in Figure 2. The Registration phase is the registration process between the device and the server. This process allows the server to provide the device serial number to the device. This process initiates the device within the proposed operations framework. The File-Deduplication phase removes redundancies between the system images stored in the server. Master Plan Control Phase is the process of planning and controlling the system images that are used for booting devices controlled by the server. The Master Plan Control allows the server to determine which system image to deploy to each device at the current time. The Resource-Efficient File-Deployment Phase is a process for determining how to respond to network boot requests from devices. The Request Queue Monitoring within this phase monitors the frequency of network boot requests and determines how system images are deployed based on the situation. There are two system image deployment methods: unicast and broadcast. When using broadcast transmission, the Block-Deployment Scheduling algorithm determines the deployment order for deduplicated system image blocks. This aim of this algorithm is to minimize the average time spent downloading system images across all devices.

The File Deployment protocol consists of a request and response phase. In the request phase, the device requests a network boot from the server. In the response phase, the server deploys the system image by either unicast or broadcast transmission, as determined by the Request Queue Monitoring.

The File-Recovery Algorithm is on the device side and recovers the deduplicated system image blocks to the original system image. The device boots from the system image recovered during this process. This algorithm is described in more detail below.

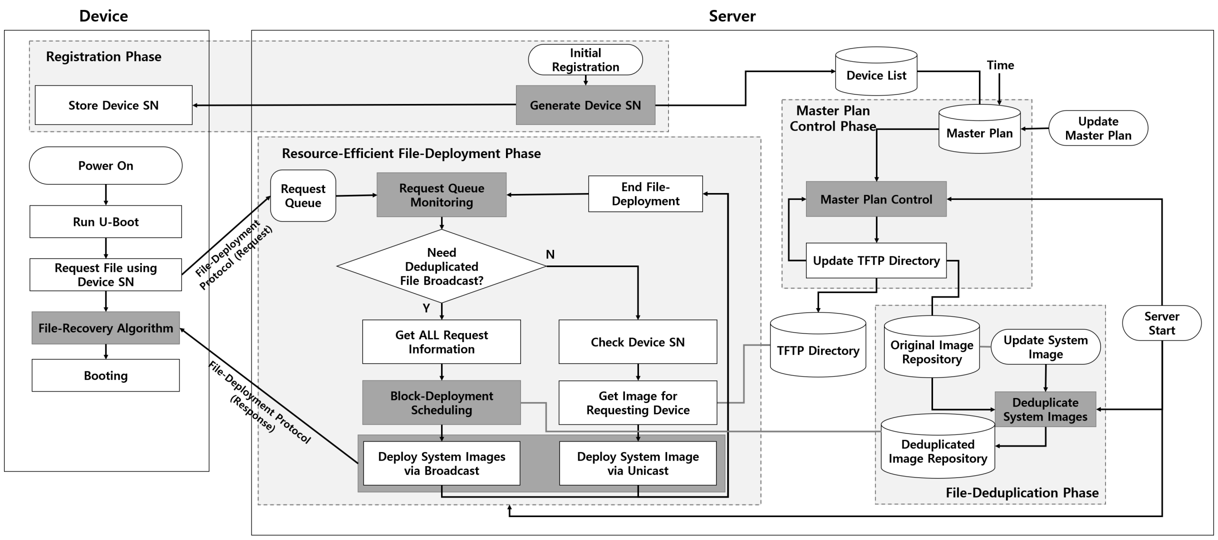

Device
Server
Registration Phase
Store Device SN
Initial Registration
Generate Device SN
Device List
Time
Master Plan Control Phase
Master Plan
Update Master Plan
Power On
Run U-Boot
Request File using Device SN
File-Recovery Algorithm
Booting
File-Deployment Protocol (Request)
File-Deployment Protocol (Response)
Resource-Efficient File-Deployment Phase
Request Queue
Request Queue Monitoring
End File-Deployment
Need Deduplicated File Broadcast?
N
Y
Get ALL Request Information
Check Device SN
Block-Deployment Scheduling
Get Image for Requesting Device
Deploy System Images via Broadcast
Deploy System Image via Unicast
Master Plan Control
Update TFTP Directory
TFTP Directory
Original Image Repository
Update System Image
Deduplicate System Images
Deduplicated Image Repository
File-Deduplication Phase
Server Start

Figure 2: RE-NETBOOT Overall Operation Flow Chart

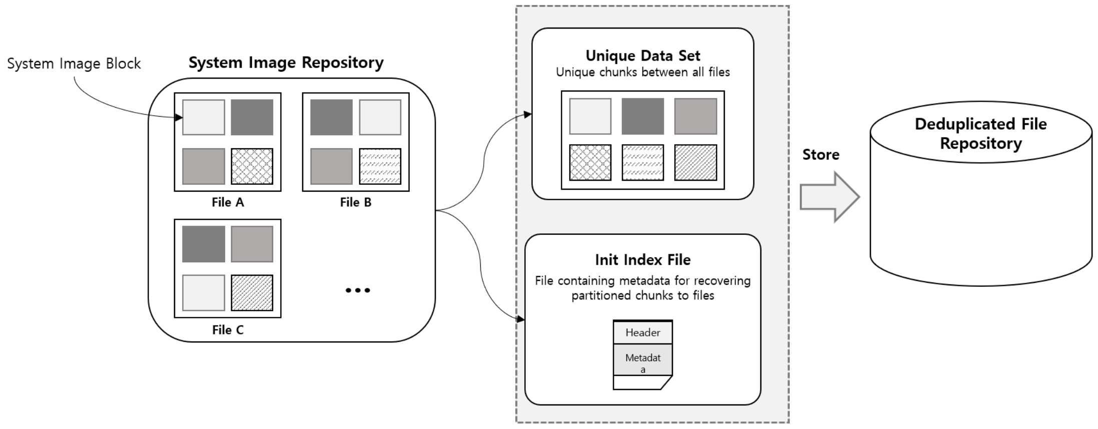

System Image Block
System Image Repository
File A
File B
File C
...
Unique Data Set
Unique chunks between all files
Init Index File
File containing metadata for recovering partitioned chunks to files
Header
Metadata
Store
Deduplicated File Repository

Figure 3: System Images Deduplication output

3.1 Registration Phase

This phase is the process of registering the device with the server for network boot. The server generates a unique Device SN for the device, passes it to the device, and the device stores it. The server manages the list of devices controlled by the server by storing the created Device SN in the device list repository. The Device SN is used when a device asks the server for a network boot, allowing the server to identify the device and distribute a system image.

3.2 File-Deduplication Phase

This phase directs the removal of redundancies from system images stored on the server. The system image is divided into fixed-length chunks and deduplicated based on the system image blocks. Data deduplication is a technique that requires extensive computation and resources, which can slow server performance. However, the proposed framework does not include the deduplication process in the network boot request and response process. As a result, the deduplication process is independent of the network boot process, so the performance impact to the network boot process is small.

In the proposed framework, the chunk size for dividing system images is designed to match the data transfer size of the file transfer protocol. This design sends one system image block to one data block. Because the proposed framework is designed based on TFTP, the system image block size is set to 512 bytes. Two types of data are generated by removing redundancies from the system images, as shown in Figure 3. A unique dataset is assigned a unique value that designates the system image block after it has undergone deduplication. All system images stored on the server are compressed into a unique dataset. The Index File is used to recover the original system image based on the system image block. Index files are divided into headers and metadata, as shown in Figure 4.

3.3 Master Plan Control Phase

This phase prepares a system image for deployment to devices based on the infrastructure operations plan. The master plan is a schedule of operating system usage for each device over time. Based on the master plan, the server determines which system image to send in response to a specific device request for a network boot. The server consults the plan of each device and copies the system image(s) needed to boot the device into the TFTP directory. The device SN is added to the file name during the copy process. This allows the server to determine which system image the device will use by device SN when

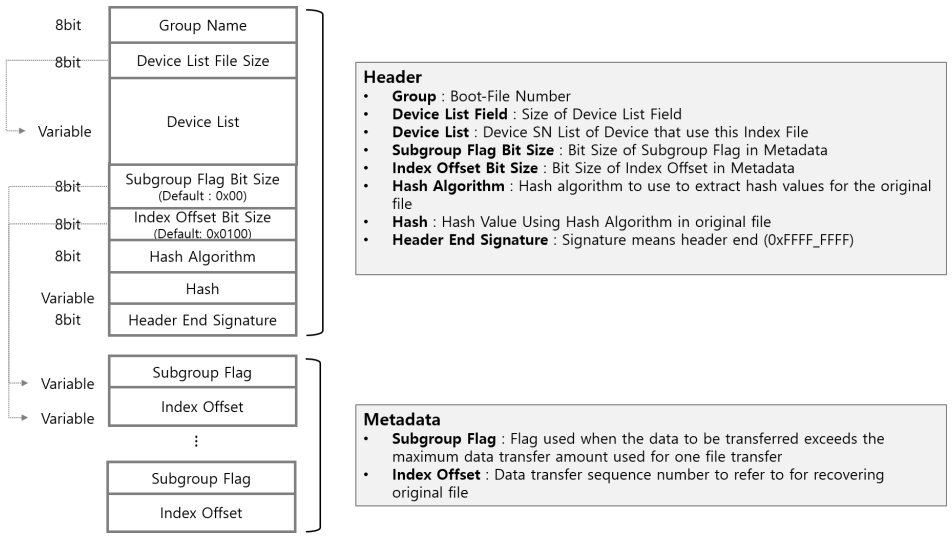

8bit
Group Name
8bit
Device List File Size
Variable
Device List
8bit
Subgroup Flag Bit Size (Default : 0x00)
8bit
Index Offset Bit Size (Default: 0x0100)
8bit
Hash Algorithm
Variable
Hash
8bit
Header End Signature
Variable
Subgroup Flag
Variable
Index Offset
Subgroup Flag
Index Offset
Header
• Group : Boot-File Number
• Device List Field : Size of Device List Field
• Device List : Device SN List of Device that use this Index File
• Subgroup Flag Bit Size : Bit Size of Subgroup Flag in Metadata
• Index Offset Bit Size : Bit Size of Index Offset in Metadata
• Hash Algorithm : Hash algorithm to use to extract hash values for the original file
• Hash : Hash Value Using Hash Algorithm in original file
• Header End Signature : Signature means header end (0xFFFF_FFFF)
Metadata
• Subgroup Flag : Flag used when the data to be transferred exceeds the maximum data transfer amount used for one file transfer
• Index Offset : Data transfer sequence number to refer to for recovering original file

Figure 4: Index File Structure

requesting a network boot. The copied system image is determined according to the master plan. This phase allows the server to dynamically change the system image that is distributed to a device.

3.4 Resource-Efficient File-Deployment Phase

This phase is a process in which the response to the actual network boot request is performed. This phase consists of two sub-processes. The first process is Request Queue Monitoring, which monitors network boot requests and uses them to determine how the server should deploy the files. By default, the server uses unicast transmission to send system images in response to network boot requests. However, if the number of simultaneous network boot requests is large, responding by unicast transmission will take time. In such situations, we designed the Request Queue Monitoring to switch the server response to broadcast transmission. The second process is Block-Deployment Scheduling, which is active when files are being deployed by broadcast transmission. When the server deploys system images by broadcast, it sends system image blocks to multiple devices at once. In this case, the time required to recover the system image depends on the deployment order of each system image block. If a device needs the last system image block, the system image cannot be recovered until the deployment of the last system image block is complete. We designed the Block-Deployment Scheduling algorithm to minimize the average time spent restoring all system images during broadcast transmission. Both processes are designed to spend the least amount of time distributing files in response to network boot requests. This is explained in more detail below.

3.4.1 Request Queue Monitoring

The proposed framework places a lightweight gateway on the server side to process through the queue for network boot requests. The Request Queue Monitoring determines how files are deployed by measuring the frequency at which network booting requests are entering the queue. By default, the response to network boot requests is unicast transmission. If a new request comes in before the server responds, queued requests will gradually accumulate. In such situations, the server switches to broadcast transmission to respond to the device requests. Each response method is shown in Figure 5.

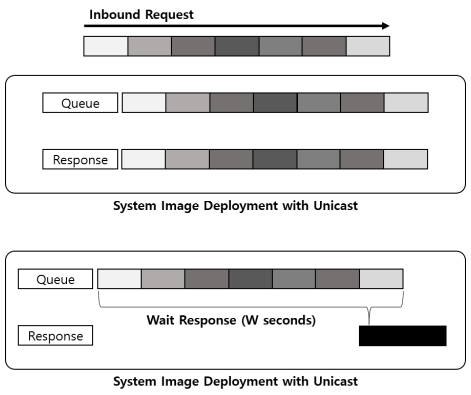

Inbound Request
Queue
Response
System Image Deployment with Unicast
Queue
Wait Response (W seconds)
Response
System Image Deployment with Unicast

Figure 5: System Image Deployment with Unicast and Broadcast

3.4.2 Block-Deployment Scheduling

This algorithm determines the order in which system image blocks are sent in response with a broadcast transmission. The goal of this scheduling is to minimize the average time spent restoring all system images by scheduling the deployment order of system image blocks. We propose two techniques for this as shown in Figure 6. The two scheduling techniques are identical in that the same system image block is not placed more than once.

The first technique is smallest-file-first (SFF) scheduling. This scheduling places the smallest system image first. The system image size is measured after deduplication within one system image. The size is also based on the number of system image blocks. In the case of SFF, the smallest system image is sent first, an approach which is like the shortest-job-first (SJF) scheduling in CPU scheduling.

The second technique is minimum-extra-first (MEF) scheduling. This scheduling places the system

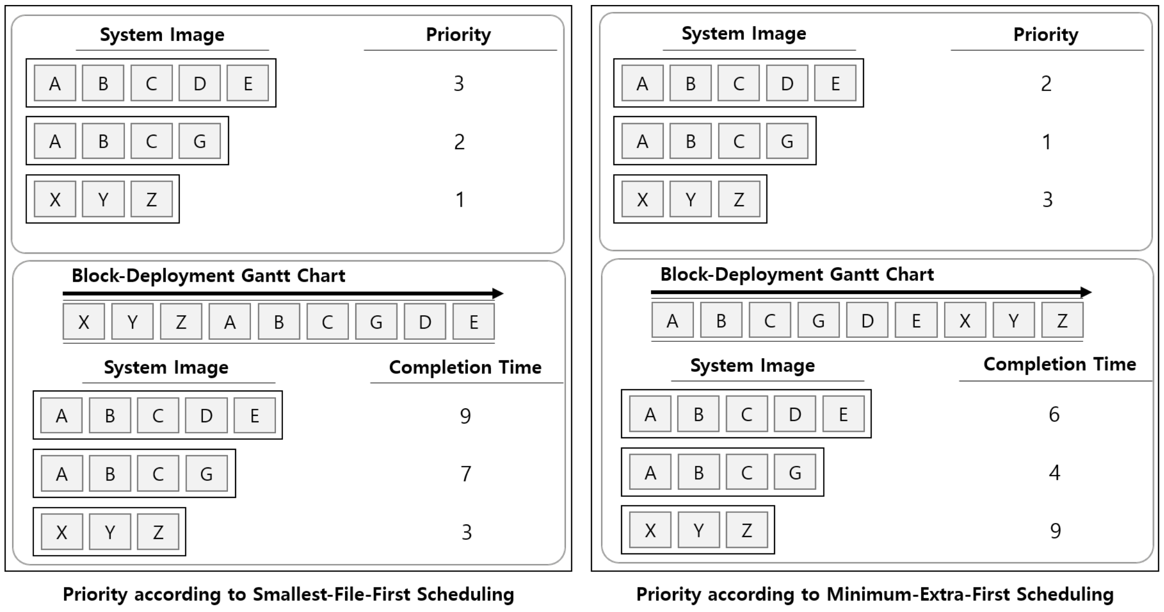


Figure 6: Block-Deployment Scheduling using Smallest-File-First (SFF) and Minimum-Extra-First (MEF)

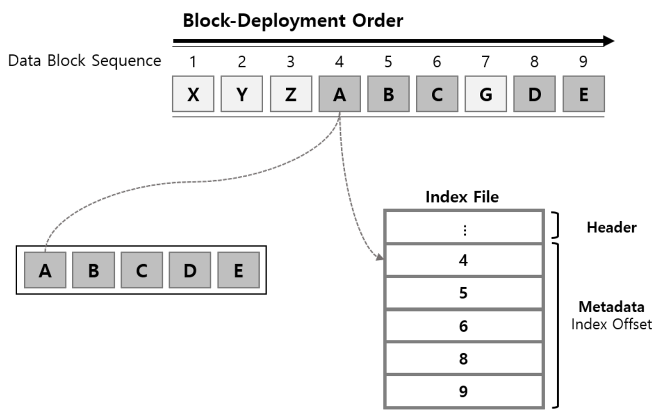


Figure 7: Example of Matching between Data Block Sequence and Index Offset

image with the least extra blocks first. Extra blocks are blocks that do not duplicate one system image with another. Fewer extra blocks for certain system images means that the system image has a lot of redundancy. In MEF scheduling, during a series of system image deployments, previously deployed system images contain most of the system image blocks of the new system image. Therefore, the system image that is sent first with MEF can be larger than that sent first with SFF, but the number of system image blocks for deploying the subsequent system images is smaller.

Once the order of block distribution is determined by scheduling, the Index File must be reconstructed. First, the Device SN that will use the system image is recorded in the Device List File of the Index File. Also, the Index Offset and Block Deployment order of the metadata must match, as shown in Figure 7. The block deployment sequence becomes the data sequence number of the file transfer protocol. The Index File records which devices will recover which system image blocks that are distributed in a series. Based on this, the device uses a file-recovery algorithm to recover the system image for booting.

3.5 File-Deployment Protocol

There are two major file transfer protocols. This is a request process in which the device makes a network boot request to the server and a process is initiated that results in the server responding to the request. The device replies with a Read Request including its Device SN to server. The device records and sends its Device SN instead of the file name requested in the “FileName” field of the TFTP Read Request. The original TFTP server sends the file corresponding to the “FileName” of the Read Request to the device. However, the proposed server-side framework selects and distributes files set through the Master Plan Control Phase based on the Device SN.

Responses from the server are subdivided into unicast and broadcast methods, the choice of which depends on the file distribution method determined by the Request Queue Monitoring. Unicast is the same as the traditional TFTP unicast response. In the case of broadcast responses, there are three phases of file distribution. We perform file distribution based on the Intel TFTP Subnet Broadcast [6], as shown in Figure 8. The first is the Negotiation Phase for distribution by broadcast. In this process, the server selects the broadcast port and master device to distribute the file. The master device performs the acknowledgment (ACK) according to the data transmission of the server, and other devices do not perform the ACK according to the data transmission of the server. The Index Deployment Phase deploys index files. For each Index File distributed, the device checks whether its Device SN is included in the Device List Field, and if so, the device saves the corresponding Index File. In the Block-Deployment Phase, system image blocks are transferred according to the block deployment order determined by the

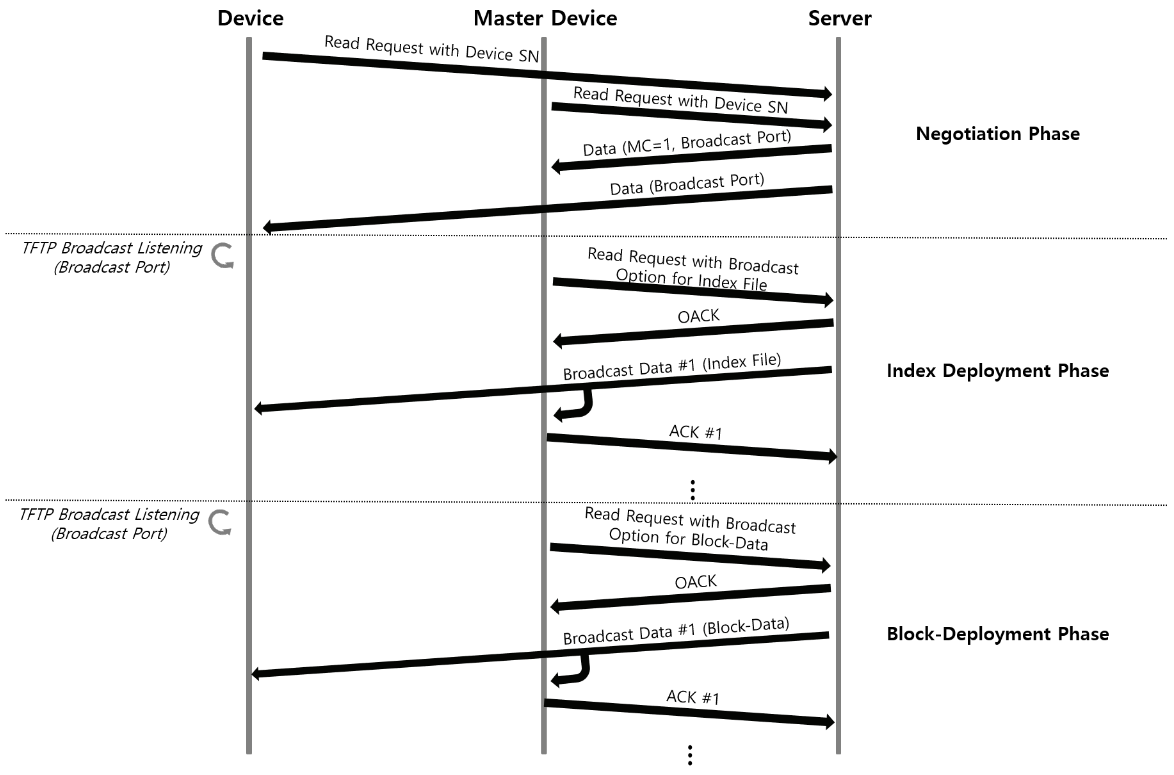

Device
Master Device
Server
Read Request with Device SN
Read Request with Device SN
Data (MC=1, Broadcast Port)
Data (Broadcast Port)
Negotiation Phase
TFTP Broadcast Listening (Broadcast Port)
Read Request with Broadcast Option for Index File
OACK
Broadcast Data #1 (Index File)
Index Deployment Phase
ACK #1
TFTP Broadcast Listening (Broadcast Port)
Read Request with Broadcast Option for Block-Data
OACK
Broadcast Data #1 (Block-Data)
Block-Deployment Phase
ACK #1

Figure 8: File-Deployment Protocol with Broadcast

Block-Deployment Scheduling algorithm.

3.6 File-Recovery Algorithm

This algorithm is used to recover the system image for booting as shown in Algorithm 1. The Index Offset of the Index File refers to the sequence number of the TFTP Data, and the algorithm places the data corresponding to that sequence number in the specified location. This action restores the deduplicated system image to the original system image and uses it to boot the device.

4 Testbed Construction

This section describes how to build a testbed. We plan to build an infrastructure of multiple embedded devices for this experiment. The server implements a module for each proposed phase. We connect multiple embedded devices and servers. We then generate a situation where the embedded devices constantly send network boot requests. Based on this testbed, we compare the proposed framework with network booting based on conventional TFTP. The framework measures the performance of scheduling techniques based on compression ratios for the deduplication and server-side overhead that occur during file distribution. On the device side, the framework measures the additional overhead incurred during the boot process. Based on this testing approach, we plan to improve the performance of the proposed framework.

Algorithm 1: File-Recovery Algorithm

```
input: Index File , File Transfer Data Block
begin
    SysImg ← Open Empty File;
    IdxFile ← Open Index File;
    while Receive Data Block do
        if $DataSequenceNumber \in IdxFile.IndexOffset$ then
            Fill SysImg;
        end
    end
    Get Hash Value in Index File;
    HV ← Hash(sysImg);
    if $HashValue == HV$ then
        DO BOOT;
    else
        DO BOOT Halt;
    end
end
```

5 Conclusion

To improve network booting, we propose RE-NETBOOT, resource-efficient network boot with a deduplication. In the proposed RE-NETBOOT, when multiple devices simultaneously request a network boot, the server removes redundancy, passing a split file to the device, and each device reassembles the boot file. The file distribution method can be changed based on the frequency of network boot requests to minimize response completion rates. A file distribution method is adopted that accelerates the response time by switching between unicast and broadcast according to the number of boot requests in the queue. In addition, when using broadcast transmission, a scheduling algorithm determines the order of deployment for system image blocks, minimizing the time it takes for a system image to be distributed to all devices. This can accelerate processing for network booting that happens in large clusters. This paper used the default method to eliminate redundancy in boot files. Future work will design a deduplication algorithm that is optimized for boot file redundancy. We also plan to address security issues using TFTP. We expect the proposed RE-NETBOOT to be used for efficient infrastructure management via network booting in an infrastructure consisting of numerous IoT devices.

Acknowledgments

This work was supported by the Institute for Information Communications Technology Promotion (IITP) of the Korea government (MSIT) [Grant No. 2019-0-00426] and by the National Research Foundation of Korea (NRF) [Grant No. NRF-2017R1C1B2003957].

References

[1] S. E. Bailey and J. W. Fake Jr. Method and system for trivial file transfer protocol (tftp) subnet broadcast, Feb. 6 2001. US Patent 6,185,623.

[2] A. Emberson. Tftp multicast option. 1997.

[3] Q. He, Z. Li, and X. Zhang. Data deduplication techniques. In *2010 International Conference on Future Information Technology and Management Engineering*, volume 1, pages 430–433. IEEE, 2010.

[4] B. Hong, D. Plantenberg, D. D. Long, and M. Sivan-Zimet. Duplicate data elimination in a san file system. In *MSST*, pages 301–314, 2004.

[5] M. Hung. Leading the iot. Technical report, Gartner, 2017.

[6] IBM. System i networking trivial file transfer protocol. Technical report, IBM, 2008.

[7] iPXE Project. iPXE-OPEN SOURCE BOOT FIRMWARE. `https://ipxe.org/`, 2015. [Online; accessed 19-August-2019].

[8] N. Mandagere, P. Zhou, M. A. Smith, and S. Uttamchandani. Demystifying data deduplication. In *Proceedings of the ACM/IFIP/USENIX Middleware'08 Conference Companion*, pages 12–17. ACM, 2008.

[9] J. Min, D. Yoon, and Y. Won. Efficient deduplication techniques for modern backup operation. *IEEE Transactions on Computers*, 60(6):824–840, 2010.

[10] A. Muthitacharoen, B. Chen, and D. Mazieres. A low-bandwidth network file system. In *ACM SIGOPS Operating Systems Review*, volume 35, pages 174–187. ACM, 2001.

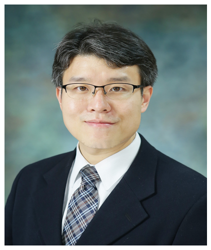
[11] W. V. A. Oliveros, C. A. M. Festin, and R. M. Ocampo. A p2p network booting scheme using a bittorrent-like protocol. In *2013 International Conference on IT Convergence and Security (ICITCS)*, pages 1–4. IEEE, 2013.

[12] K. Sollins. The tftp protocol (revision 2). 1992.

[13] S. Takada, A. Sato, Y. Shinjo, H. Nakai, A. Sugiki, and K. Itano. A p2p approach to scalable network-booting. In *2012 Third International Conference on Networking and Computing*, pages 201–207. IEEE, 2012.

Author Biography

Keon-Ho Park received the B.S. degrees in information security from Sejong University, Korea, in 2017. He is currently an M.S. at Graduate school of Sejong University. His research interests include information security, network, embedded machine, and trust Architecture.



Ki-Woong Park received the B.S. degree in computer science from Yonsei University, South Korea, in 2005, and the M.S. and Ph.D. degrees in electrical engineering from the Korea Advanced Institute of Science and Technology in 2007 and 2012, respectively. He was a Senior Researcher with the National Security Research Institute. He is currently a Professor with the Department of Computer and Information Security, Sejong University. His research interests include security issues for cloud and mobile computing systems as well as the actual system implementation and subsequent evaluation in a real computing system. He was a recipient of the 2009-2010 Microsoft Graduate Research Fellowship.