

Boot Loader

구조설계서

2022-05-01

홍길동

이 문서는 소프트웨어 구조 설계자 양성 프로그램에서 양성 인력이 작성한 **Boot Loader** 개발을 위한 구조 설계서입니다.



(주) 보이는 소프트웨어 연구소
조용진(drajin.cho@bosornd.com)

REVISION HISTORY

Version	Date	Author	Description
0.1	2022-04-04	홍길동	초기 문서 생성
0.2	2022-04-11	홍길동	품질 시나리오 Revision
0.3	2022-04-17	홍길동	후보 구조 분석 추가
0.4	2022-04-25	홍길동	최종 구조 설계 Revision
0.5	2022-05-01	홍길동	시스템 구조 기술

1. 시스템 정의	6
1.1. 시스템 개요	6
1.2. 시스템 정의	7
1.2.1. BIOS	8
1.2.2. User	8
1.2.3. Kernel	8
1.2.4. DRAM	8
1.2.5. I/O Device	8
1.2.6. Storage	9
1.2.7. 시스템 경계	9
1.3. 시스템 제약 사항	9
1.4. 비즈니스 드라이버	10
2. 요구사항	11
2.1. 기능적 요구사항	11
2.1.1. UC_01 커널 이미지 Load 및 수행	11
2.1.2. UC_02 패스워드 입력	13
2.1.3. UC_03 Shell 수행	13
2.1.4. UC_04 Boot 환경 설정	15
2.2. 비기능적 요구사항	16
2.2.1. NFR_01 Boot Loader 수행 시간	16
2.3. 품질 속성	16

2.3.1. QA_01 Boot Loader 수행 시간	16
2.3.2. QA_02 디바이스 추가 및 변경 용이성.....	17
2.3.3. QA_03 Shell Command 처리 성능.....	17
2.3.4. QA_04 커널 이미지 로딩 가능성	17
커널 이미지 로딩 가능성	17
2.3.5. QA_05 Shell Command 추가 및 확장 용이성	18
3. 시스템 구조	19
3.1. UC_01에 대한 동작 상세.....	20
3.1.1. Config 설정	20
3.1.2. Kernel Image 로드	21
3.1.3. Kernel 변조 체크	22
3.1.4. Shell Mode 진입 대기	22
3.1.5. Kernel 수행.....	23
3.2. UC_02에 대한 동작 상세.....	23
3.2.1. 기존 암호 읽기	24
3.2.2. 사용자 암호 입력 및 일치 확인.....	24
3.3. UC_03에 대한 동작 상세.....	25
3.3.1. Shell Mode 진입 및 커맨드 입력	26
3.3.2. Shell Command 수행.....	27
4. 모듈 사양	28
4.1. Device Layer.....	29

4.1.1. Storage Package.....	29
4.1.2. IODevice Package	29
4.2. Core Layer.....	30
4.2.1. Core Management Package	30
4.2.2. File System Package	31
4.2.3. Log Package.....	31
4.2.4. CLI Package	31
4.3. Extension Layer.....	31
4.3.1. Kernel Package.....	32
4.3.2. Shell Package.....	32
4.3.3. Config Package	33
4.3.4. Password Package	33
4.4. Business Layer.....	33
4.4.1. Boot Loader Package	34
부록	35
A. 도메인 모델.....	43
B. 품질 시나리오	47
C. 품질 시나리오 분석	50
D. 후보 구조.....	54
E. 후보 구조 평가.....	100

F. 최종 구조 설계 121

1. 시스템 정의

1.1. 시스템 개요

Boot Loader (Boot Loader)란 운영체제가 시동되기 이전에 미리 실행되면서 커널이 올바르게 시동되기 위해 필요한 모든 관련 작업을 마무리하고 최종적으로 운영체제를 구동시키기 위한 목적을 가진 프로그램이다.

아래 그림 1은 부트 시퀀스를 나타낸다. 컴퓨터 전원이 켜지면, ROM (Read-only memory)에 들어 있는 BIOS가 로드 된다. BIOS는 컴퓨터에 연결된 저장 매체에서 설정된 부팅 순서대로 Boot Loader를 불러오게 된다. 이 때, 하드 디스크가 첫 번째 부팅 장치로 설정되어 있으면, BIOS는 하드 디스크의 MBR (Master Boot Record) 영역에 저장된 Boot Loader를 로드한다. Boot Loader는 커널을 시동시키기 전 필요한 하드웨어들을 준비하고, 커널 이미지를 메모리로 로드 한다.

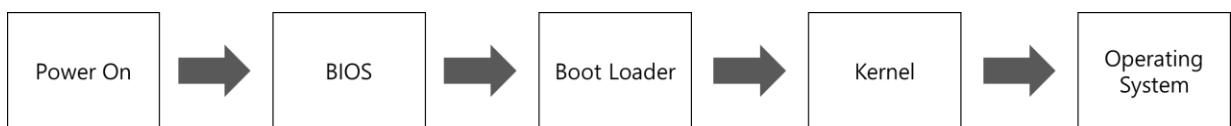


그림 1. 부트 시퀀스.

본 문서에서는 데이터센터 서버에 설치되는 Boot Loader의 소프트웨어 구성 항목의 구조적 설계를 다룬다. 아래 그림 2는 데이터센터에서 널리 사용되는 서버 구조를 나타낸다. 데이터센터용 서버에는 많은 수의 CPU 코어 (약 50개 ~ 100개), DRAM, GPU, NIC, HDD/SSD 등 다양한 컴포넌트들이 장착되어 있다. 주로 Windows나 Linux가 설치되면, 커널 수행을 위해 필요한 동작들이 많다. 따라서 Embedded 환경과 비교하여 BIOS와 Boot Loader도 다양한 기능을 제공하고 있다.



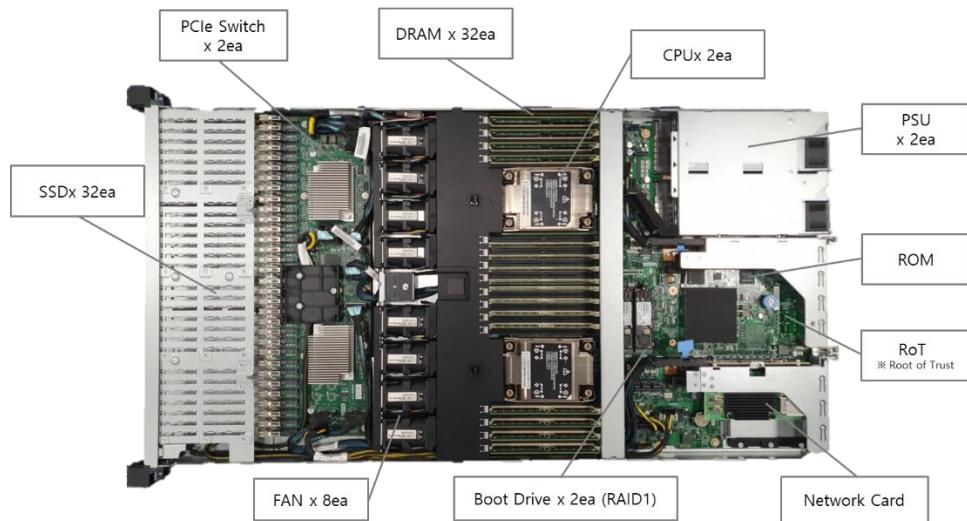


그림 2. 서버 구조.

아래 그림 3은 데이터센터 서버에서 사용되는 Boot Loader 컴포넌트 구성 및 역할을 나타낸다. Boot Loader는 Storage의 MBR (Master Boot Record) 영역에 존재하며, 파일 시스템 상에 존재하는 Kernel Image를 DRAM으로 로드 해주는 역할을 담당한다. 이를 위해, File System 및 필요한 Device Driver를 포함하고 있다. 추가로, 사용자에게 시스템 점검 및 부팅 환경 설정을 위한 Shell 기능도 제공한다.

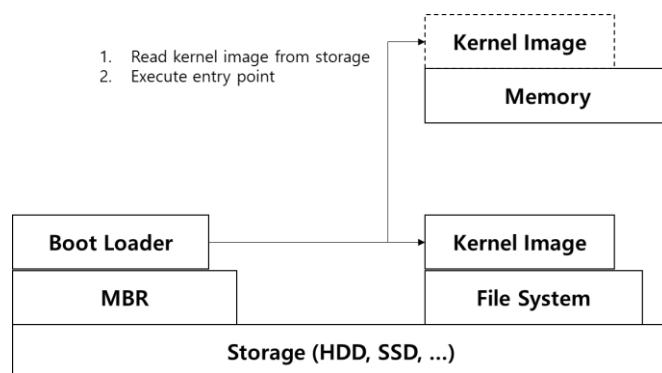


그림 3. Boot Loader 동작.

1.2. 시스템 정의

본 문서에서 정의하는 Boot Loader 개념 및 개발 범위는 그림 4와 같다.

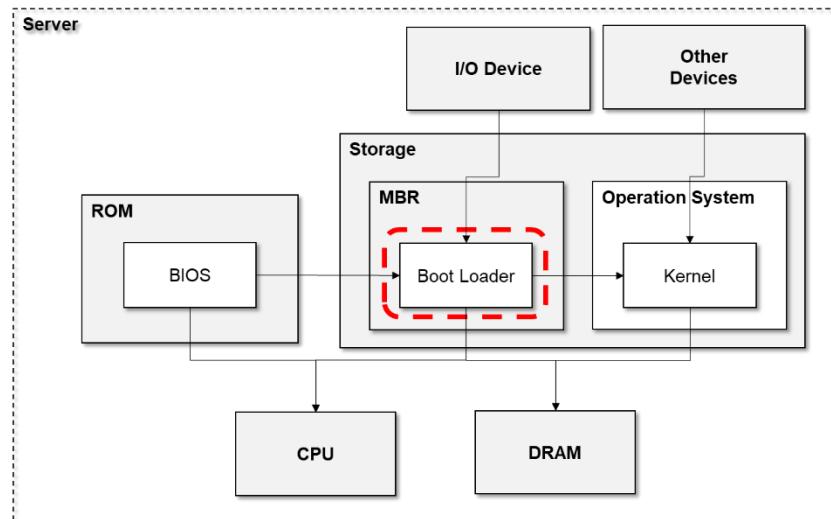


그림 4. 시스템 Layer 및 개발 범위.

1.2.1. BIOS

BIOS는 서버 전원이 켜지면 CPU가 처음으로 수행하는 소프트웨어이다. CPU, DRAM, Disk에 대한 POST (Power-on-Self-Test)를 수행한 후, MBR에 저장된 Boot Loader를 DRAM에 로드한 후, 이를 수행한다.

1.2.2. User

User는 서버 사용자이며, Boot Loader Configuration 설정, Kernel Image를 제공하며, Shell Command를 통해 Boot Loader가 제공하는 명령어를 입력한다.

1.2.3. Kernel

Kernel은 운영 체제의 핵심이 되는 소프트웨어로 스토리지에 저장되어 있다가, Boot Loader로부터 수행된다. Kernel은 (1) 메모리 관리, (2) 프로세스 관리, (3) 장치 드라이버 관리, (4) 시스템 호출 및 보안을 담당한다.

1.2.4. DRAM

휘발성 메모리로 Kernel Image 수행을 위해 로드하는 용도로 사용된다.

1.2.5. I/O Device

Keynote, 모니터 등 사용자와의 입출력을 위한 주변 장치를 의미한다.

1.2.6. Storage

비휘발성 메모리이다. Boot Loader나 Kernel Image를 수행하기 위한 공간으로 사용한다.

1.2.7. 시스템 경계

Boot Loader는 BIOS로부터 DRAM에 로드 되어, 스토리지에 저장된 Kernel Image를 DRAM으로 로드하고, 이를 호출한다. 또한, Boot Loader는 사용자로부터 I/O Device를 통해 필요한 설정 값 및 명령어를 입력 받아, 이를 처리하고, 그 결과를 보여준다.

따라서 본 문서에서 설계하고자 하는 Boot Loader 경계는 아래 그림과 같다.

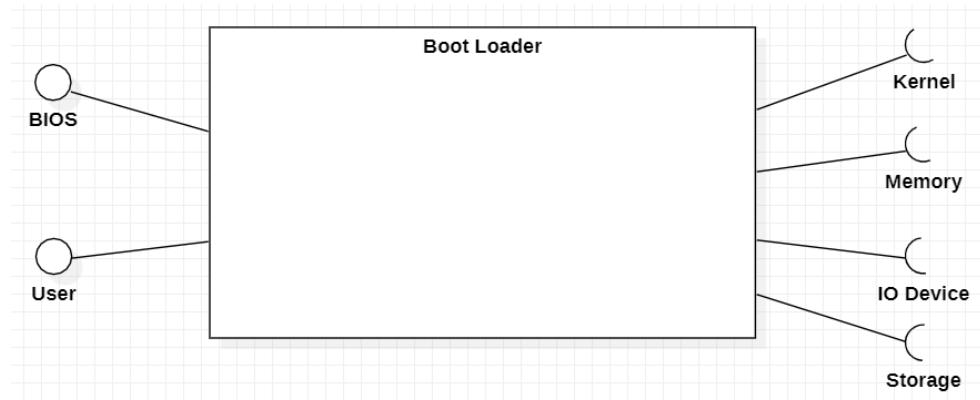


그림 5. 시스템 경계.

1.3. 시스템 제약 사항

- 서버용 보드에 설치된 BIOS는 Embedded용 BIOS와 달리 CPU, DRAM, Disk에 대한 POST (Power-on-Self-Test)를 수행한다. 따라서 Boot Loader에서는 Kernel Image 로드 성능을 위해 추가적인 POST를 수행하지 않는다.
- Kernel Image는 파일 시스템의 파일 형태로 저장된다. Ext2, Ext3, Ext4, XFS를 지원하여야 하며, 파일 시스템 UUID로 판별하여야 한다.
- 서버에 장착된 IO Device는 Serial Port 장치로 제한한다. 데이터센터의 특성 상 사용자가 직접 서버에 접근하는 경우가 매우 드물며, 따라서 가장 간단한 형태의 IO Device만 장착된다.

1.4. 비즈니스 드라이버

Boot Loader 설계를 위해 고려해야 할 주요 비즈니스 드라이버는 아래와 같다. 비즈니스 드라이버는, 본 과제의 구조설계에 있어서 중요한 품질이 무엇인가를 판단하는데 중요한 척도로 사용된다.

- Boot Loader는 최대한 빨리 부팅 작업을 완료하여, 사용자의 만족도를 높여야 한다.
- 개발할 Boot Loader는 디바이스 장치 추가 및 변경에 대해 신속하게 대응하여 원하는 기간 내에 제품 개발을 완료할 수 있어야 한다.
- Booting 과정에서 발생하는 장애 상황으로 인한 사용자의 불편은 최소화되어야 한다.
- 일부 사용자의 악의적인 공격에 대한 대비가 충분히 되어있어야 한다.

2. 요구사항

2.1. 기능적 요구사항

본 시스템의 Use Case Diagram은 아래 그림과 같다.

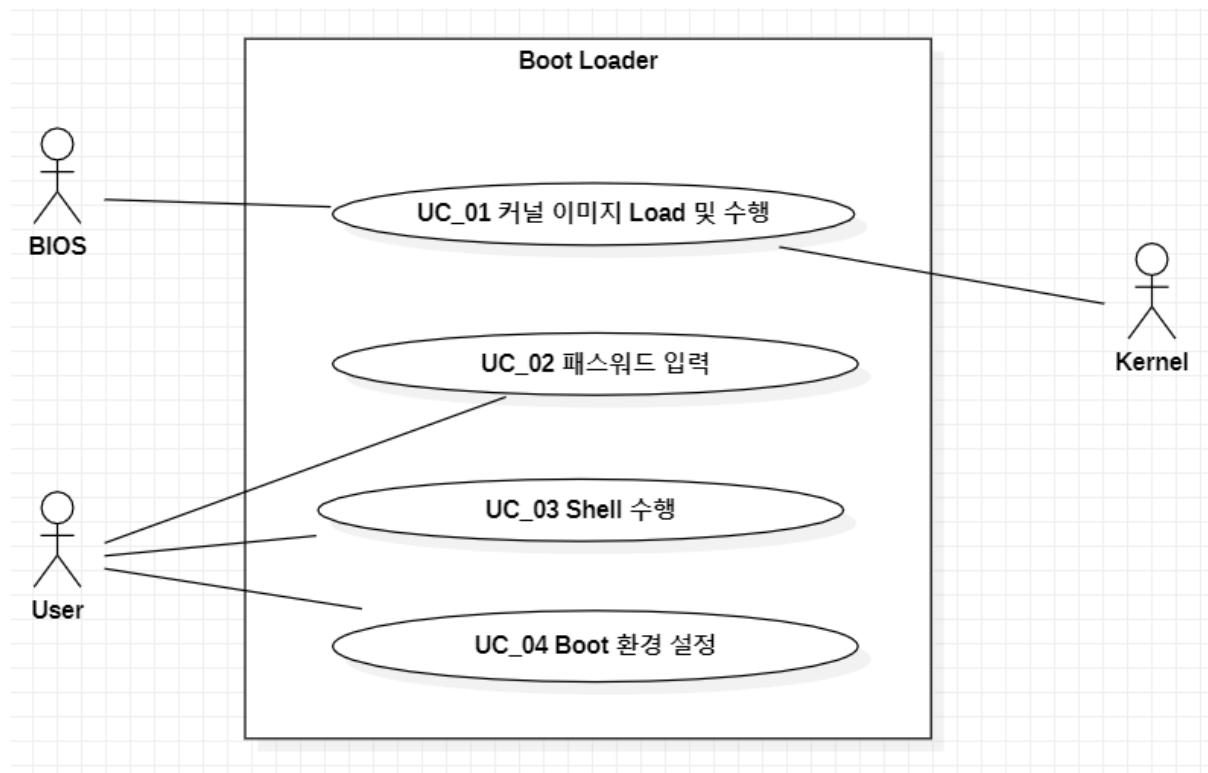


그림 6. Use Case Diagram.

2.1.1. UC_01 커널 이미지 Load 및 수행

UC_01	커널 이미지 Load 및 수행
설명	Boot Loader는 Booting Device에 저장된 커널 이미지를 DRAM에 로드하고, 수행한다.
행위자	BIOS
선행조건	Boot Loader 로드를 성공적으로 완료한 상태
후행조건	커널 이미지가 실행된다.

기본 동작	<ol style="list-style-type: none"> 1. BIOS 는 Boot Loader 를 수행한다. 2. Boot Loader 는 Boot Config 파일로부터 수행에 필요한 설정 값을 세팅한다. 3. Boot Loader 는 파일 시스템에서 커널 이미지 파일을 읽어와 DRAM 에 로드 한다. 4. Boot Loader 는 커널 이미지 Signature 확인을 통해 커널 이미지 변조 여부를 체크한다. 5. Boot Loader 는 Shell Mode 진입 대기 시간 (10s) 동안 대기한다. 6. Boot Loader 는 커널 이미지를 수행한다.
추가 동작	<p>AF1. 사용자 패스워드 입력 분기점: 기본 동작 2</p> <ol style="list-style-type: none"> 1. 사용자 패스워드가 설정되어 있을 경우, Boot Loader 는 'UC_02 패스워드 입력'을 우선 수행한 후, 기본 동작 2로 돌아간다.
	<p>AF2. Boot Config 파일 미존재 분기점: 기본 동작 2</p> <ol style="list-style-type: none"> 1. 미리 생성한 Boot Config 파일이 존재하지 않을 경우, Boot Loader 는 Default 설정 값을 읽어온 후, 기본 동작 3으로 돌아간다.
	<p>AF3. Kernel Image 미존재 분기점: 기본 동작 3</p> <ol style="list-style-type: none"> 1. 만약 Kernel Image 가 존재하지 않는다면, Boot Loader 는 "Kernel Image 존재하지 않습니다."와 같은 적절한 에러 메시지를 출력한 후, 본 Use Case 를 종료한다.
	<p>AF4. 커널 이미지 변조 발생 분기점: 기본 동작 4</p> <ol style="list-style-type: none"> 1. Kernel Image 가 변조되었다면, Boot Loader 는 "Kernel Image 가 올바르지 않습니다."와 같은 적절한 에러 메시지를 출력한 후, 본 Use Case 를 종료한다.
	<p>AF5. Shell Mode 진입 분기점: 기본 동작 5</p> <ol style="list-style-type: none"> 1. 대기 시간 동안 User 가 Shell Mode 진입 키 (ex. F12)가 입력하였다면, "UC_03 Shell 수행'을 수행한 후, 본 Use Case 를 종료한다.

2.1.2. UC_02 패스워드 입력

UC_02	패스워드 입력
설명	User는 보안성 _향_상_을 위해, Boot Loader 수행 패스워드를 설정할 수 있다.
행위자	User
선행조건	- 'UC_01 Boot Loader 수행' 2단계까지 수행된 상태 - 사용자 패스워드가 설정된 상태
후행조건	'UC_01 Boot Loader 수행' 3단계부터 수행 재개된다.
기본 동작	<ol style="list-style-type: none"> 1. Boot Loader 는 User 가 패스워드를 입력할 때까지 대기한다. 2. User 는 미리 설정한 패스워드를 입력한다. 3. Boot Loader 는 User 가 기 설정한 패스워드를 MBR 로부터 읽어온다. 4. Boot Loader 는 읽어온 패스워드를 복호화 한다. 5. Boot Loader 는 User 가 입력한 패스워드와 기 설정한 패스워드가 일치하는지를 체크한 후, 일치한다면 본 Use Case 를 종료한다.
추가 동작	<p>AF1. 잘못된 패스워드 입력 분기점: 기본 동작 3</p> <ol style="list-style-type: none"> 1. User 가 잘못된 패스워드를 입력할 경우, Boot Loader 는 "잘못된 패스워드입니다."와 같은 적절한 에러 메시지를 출력한 후, 1 단계를 수행한다. <p>AF2. 잘못된 패스워드 입력 분기점: 기본 동작 3</p> <ol style="list-style-type: none"> 1. User 가 5 회 이상 잘못된 패스워드를 입력할 경우, Boot Loader 는 "잘못된 패스워드입니다. 더 이상 부팅을 진행할 수 없습니다."와 같은 적절한 에러 메시지를 출력한 후, 부팅 과정을 종료한다.

2.1.3. UC_03 Shell 수행

UC_03	Shell 수행
설명	User는 시스템 점검 및 부팅 환경 설정을 위해 Shell을 수행한다.
행위자	User
선행조건	'UC_01. 커널 이미지 Load 및 수행' 2단계까지 수행된 상태
후행조건	
기본 동작	<ol style="list-style-type: none"> 1. User 는 Boot Loader 수행 과정에서 Shell Mode 진입 키 (ex. F12)를

	<p>입력한다.</p> <ol style="list-style-type: none"> 2. Boot Loader 는 Shell Mode 수행한다. 3. Boot Loader 는 User 가 명령어를 입력할 때까지 대기한다. 4. User 는 미리 정의된 명령어 (아래 참조)를 입력한다. <p>[부팅 관련]</p> <ul style="list-style-type: none"> - root: 입력한 장치를 root device 로 지정 - blocklist: 입력한 파일이 저장된 blocklist 를 확인 - kernel: 부팅에 사용할 커널 이미지 파일 경로를 지정 - boot: 지정된 커널로 부팅 - reboot: 시스템을 재부팅 시키는 명령어 - halt: 시스템을 정지시키는 명령어 <p>[파일 관련]</p> <ul style="list-style-type: none"> - configfile: 지정 한 파일로부터 설정을 로드하는 명령어 - cat: 지정한 파일 내용 확인 - find: 지정한 파일이 위치한 장치명을 찾아주는 명령어 <p>[장치 관련]</p> <ul style="list-style-type: none"> - displayapm: APM (Advanced Power Management) BIOS 정보 출력 - displaymem: 물리적으로 DRAM 이 설치되어 있는 시스템 주소 공간에 대한 map 표시 - geometry: 지정된 device 에 대한 정보 출력 <p>[기타]</p> <ul style="list-style-type: none"> - help: Shell 명령어들에 대한 도움말 출력 - clear: 화면을 지우는 명령어 - md5crypt: Boot Loader 패스워드 설정을 위한 MD5 암호 문자 생성기 - quit: Shell Mode 종료 커맨드 <ol style="list-style-type: none"> 5. Boot Loader 는 User 가 입력한 커맨드의 Valid 여부를 검증한다. 6. Boot Loader 는 커맨드를 처리한다. 각 명령어별 상세 처리 과정은 아래와 같다. <ul style="list-style-type: none"> - 부팅 관련 명령: Boot Loader 는 커널 이미지가 저장된 hard disk 를 access 하여, 이를 DRAM 으로 로드한 후, 커널에 제어권을 넘긴다. - 파일 관련 명령: Boot Loader 는 파일 시스템으로부터 User 가 지정한 파일을 읽은 후, 내용을 로드하거나, 출력한다. 혹은 Boot Loader 는 User 가 지정한 파일의 위치를 출력한다. - 장치 관련: Boot Loader 는 User 가 원하는 장치로부터 정보를 읽어와 이를 출력한다.
--	--

	<ul style="list-style-type: none"> - 기타: Boot Loader 는 명령어들에 대한 상세 도움말을 출력하거나, Shell 화면을 지우는 등 User 가 입력한 커맨드를 수행한다. <p>7. Boot Loader 는 커맨드 처리 결과를 출력한다.</p> <p>8. 본 Use Case 의 3 단계를 수행한다.</p>
추가 동작	<p>AF1. Invalid 커맨드 입력 분기점: 기본 동작 5</p> <p>1. User 가 입력한 명령어가 Valid 하지 않다면, Boot Loader 는 "잘못된 명령어입니다.", "Parameter 가 올바르지 않습니다." 등과 같은 적절한 에러 메시지를 출력한 후, 본 Use Case 의 3 단계를 수행한다.</p>
	<p>AF2. Quit 명령어 입력 분기점: 기본 동작 6</p> <p>1. User 가 종료 명령어 (ex. quit)을 입력하였다면, Boot Loader 는 Shell Mode 를 종료하고, 본 Use Case 를 종료한다.</p>

2.1.4. UC_04 Boot 환경 설정

UC_04	Boot 환경 설정
설명	User는 Boot Loader 수행에 필요한 설정 값을 변경할 수 있다.
행위자	User
선행조건	설정 값의 위치가 이미 정의된 상태
후행조건	User 설정 값으로 'UC_01 커널 이미지 Load 및 수행' 3단계를 수행한다.
기본 동작	<p>1. User 는 미리 정의된 위치에 Config 파일 (ex. /boot/bl/bl.conf)을 생성한다.</p> <p>2. Boot Loader 는 Config 파일로부터 설정 값을 읽어온다.</p> <p>3. Boot Loader 는 설정 값의 Valid 여부를 체크한다.</p> <p>4. 설정 값이 Valid 할 경우, Boot Loader 는 해당 설정 값으로 내부 설정 값을 업데이트 한다.</p>
추가 동작	<p>AF1. Invalid 설정 값 분기점: 기본 동작 3</p> <p>1. 설정 값이 Valid 하지 않을 경우, Boot Loader 는 "Configuration File 에 Invalid 한 설정 값이 기술되어 있습니다."와 같은 적절한 에러 메시지를 출력한 후, 본 Use Case 를 종료한다.</p>

2.2. 비기능적 요구사항

2.2.1. NFR_01 Boot Loader 수행 시간

NFR_01	Performance	Boot Loader 수행 시간
설명	Boot Loader는 10초 내에 Kernel 부팅에 필요한 모든 동작을 완료하여야 한다.	
환경	BIOS가 정상적으로 동작하는 상태	
자극	BIOS 가 Boot Loader 를 수행한다.	
반응	1. BIOS 는 Boot Loader 를 수행한다. 2. Boot Loader 는 수행에 필요한 설정 값을 세팅한다. 3. Boot Loader 는 파일 시스템에서 커널 이미지 파일을 읽어와 DRAM 에 로드 한다. 4. Boot Loader 는 커널 이미지 Signature 확인을 통해 커널 이미지 변조 여부를 체크한다 5. Boot Loader 는 커널 이미지를 수행한다.	
측정	$[부팅 시간] = [Kernel 수행 시작 시각] - [Boot Loader 수행 시작 시각]$	
제약	$[부팅 시간] \leq 10s$	

2.3. 품질 속성

2.3.1. QA_01 Boot Loader 수행 시간

QA_01	Performance	Boot Loader 수행 시간
설명	Boot Loader의 수행 시간은 빠를수록 좋다.	
환경	BIOS가 정상적으로 동작하는 상태	
자극	BIOS 가 Boot Loader 를 수행한다.	
반응	1. BIOS 는 Boot Loader 를 수행한다. 2. Boot Loader 는 수행에 필요한 설정 값을 세팅한다. 3. Boot Loader 는 파일 시스템에서 커널 이미지 파일을 읽어와 DRAM 에 로드 한다. 4. Boot Loader 는 커널 이미지 Signature 확인을 통해 커널 이미지 변조 여부를 체크한다 5. Boot Loader 는 커널 이미지를 수행한다.	

측정	[부팅 시간] = [Kernel 수행 시작 (반응5)] – [Boot Loader 수행 시작 시작 (반응1)]
----	---

2.3.2. QA_02 디바이스 추가 및 변경 용이성

QA_02	Modifiability	디바이스 추가 및 변경 용이성
설명	Storage, IO Device의 다양한 디바이스가 추가되거나, 이미 장착된 디바이스 드라이버가 변경되었을 때, 시스템 변경을 위한 개발 비용이 최소화되어야 한다.	
환경	개발 완료 이후 시점 혹은 해당 요구사항 구현 진행 중인 시점	
자극	디바이스 추가 및 변경에 대한 신규 요구사항 요청	
반응	요구사항을 만족하도록 Boot Loader 를 수정 개발한다.	
측정	[개발 비용 (M/M)] = [수정, 검증을 다시 해야 하는 모듈] / [파일의 크기(LoC)]	

2.3.3. QA_03 Shell Command 처리 성능

QA_03	Performance	Shell Command 수행 시간
설명	Shell Command는 빠르게 처리될수록 좋다.	
환경	Shell Mode를 수행하는 상태	
자극	User 는 수행을 원하는 Shell Command 를 입력한다.	
반응	Boot Loader 는 Shell Command 를 수행하고, User 에게 결과를 리턴한다.	
측정	[Shell Command 수행 시간] = [Shell Command 입력 시작] – [Shell Command 결과 리턴 시작]	

2.3.4. QA_04 커널 이미지 로딩 가능성

QA_04	Availability	커널 이미지 로딩 가능성
설명	Boot Loader 수행 과정에서 장애 발생 시, 정상 상태로 빨리 돌아갈 수 있어야 한다.	
환경	Boot Loader가 커널 이미지 로딩 진행 중인 상태	
자극	Boot Loader 는 수행 중 장애 상황을 발견한다.	

반응	<ol style="list-style-type: none"> 1. Boot Loader 는 커널 이미지 로딩을 수행한다. 2. Boot Loader 는 커널 이미지 로딩 과정에서 장애 상황을 발견한다. 3. Boot Loader 는 장애 복구를 시도한다. 4. Boot Loader 는 사용자에게 “부팅 과정에서 xx 장애가 발생하였지만, 정상적으로 복구하여 부팅을 진행합니다.”와 같은 적절한 메시지를 출력한 후, 부팅을 재개한다.
측정	[정상 복구 시간] = [Recover 완료 시각 (반응 4)] – [Fail 발생 시각 (반응 2)]

2.3.5. QA_05 Shell Command 추가 및 확장 용이성

QA_05	Modifiability	Shell Command 추가 및 확장 용이성
설명	Shell Command가 추가되거나 확장되었을 때, 시스템의 변경을 위한 개발 비용이 최소화되어야 한다.	
환경	개발 완료 이후 시점 혹은 해당 요구사항 구현 진행 중인 시점	
자극	Shell Command 에 대한 요구사항 변경 혹은 신규 요구사항 요청	
반응	요구사항을 만족하도록 Boot Loader 를 수정 개발한다.	
측정	[개발 비용 (M/M)] = [수정, 검증을 다시 해야 하는 모듈] / [파일의 크기(LoC)]	

3. 시스템 구조

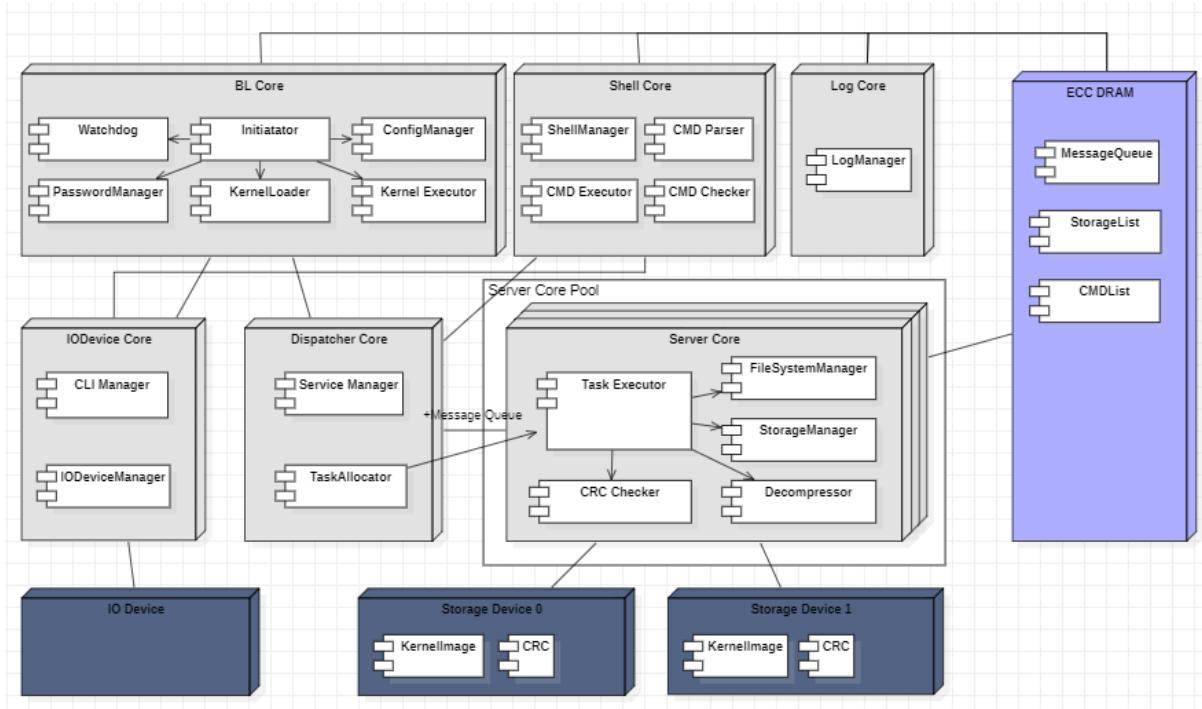


그림 7. 동작 측면에서의 Deployment View.

본 문서에서 제안하는 Boot Loader의 동작 실행 측면의 구조는 위 그림과 같다. 5개의 Dedicated Core와 Server Core Pool에 속하는 Core들로 구성된다. 각 Core의 역할을 다음과 같다.

1. BL Core: Boot Loader 핵심 서비스 로직들을 처리하는 Core이자 Dispatcher Core에게 서비스를 요청하는 Client Core.
2. Shell Core: Shell Mode 지원 및 사용자의 Shell Command 입력에 대한 처리를 위한 Core
3. Log Core: Log Message 출력을 위한 전담 Core
4. IO Device Core: IO Device 관리 전담을 위한 Core
5. Dispatcher Core: Server Core에서 수행할 서비스들을 관리하고, Client Core와 Server Core 간의 채널을 연결해주는 Core. Client Core 부하 감소를 위한 목적.
6. Server Core Pool: 동일 Task를 병렬화 수행하는 Core들의 집합. Server Core들 간의 Task 균등 분배를 위해, Server Core Pool 형태로 운영.

3.1. UC_01에 대한 동작 상세

아래 그림은 UC_01에 대한 시스템 관점에서의 Sequence Diagram이다. 본 장에서는 이를 바탕으로 각 단계에 대한 Core별 상세 동작 구조를 기술한다.

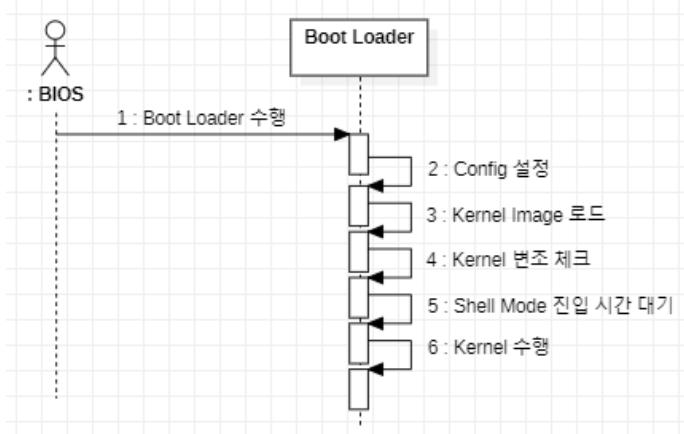


그림 8. 시스템 관점의 Kernel Load Sequence Diagram.

3.1.1. Config 설정

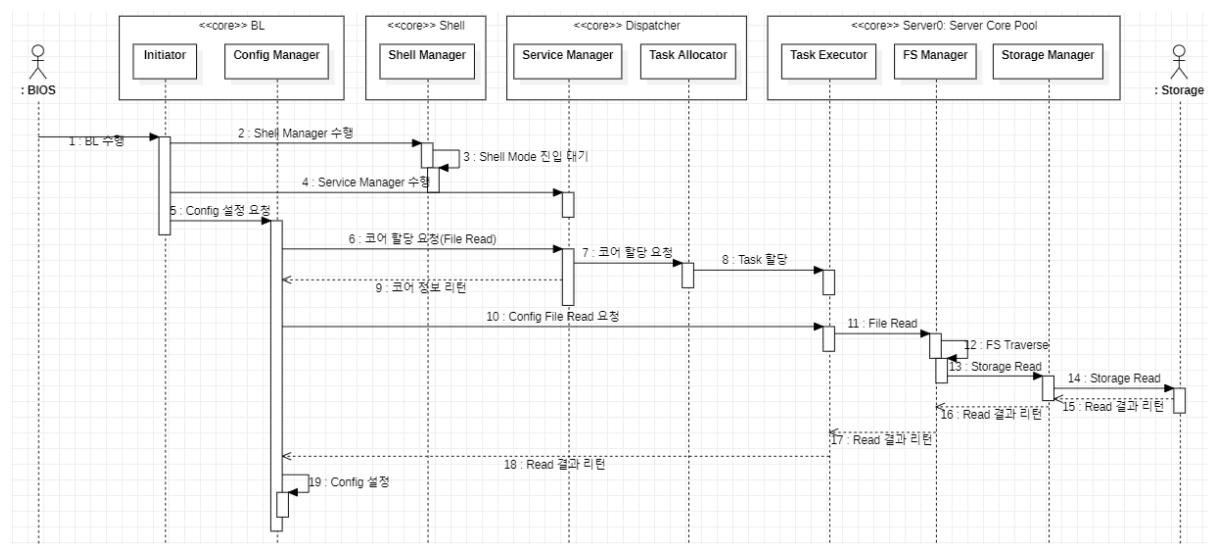


그림 9. Config 설정 Sequence Diagram.

위 그림은 Boot Loader가 수행되었을 때부터 Config 설정 완료하기까지의 Sequence Diagram을 나타낸다. Boot Loader가 수행되면, Initiator는 가장 먼저 Shell Mode 진입 시점을 앞당기기 위해, Boot Loader

Shell Manager를 수행시킨다. Shell Core는 Shell Manager 수행을 완료한 후, Shell Mode 진입 대기를 위한 타이머를 시작한다. Initiator는 Config Manager를 통해, Config 설정을 요청한다. Config Manager는 File System으로부터 Config File을 읽어 오기 위해, Dispatcher Core의 Service Manager에게 File Read 코어 할당을 요청한다. Dispatcher Core는 Server Core Pool 중 하나의 Server Core에게 관련 Task를 할당하고, 해당 정보를 Config Manager에게 알려준다. Config Manager는 할당된 Server Core에게 Task 수행을 요청하며, 결과를 넘겨 받아, Config 설정을 완료한다.

3.1.2. Kernel Image 로드

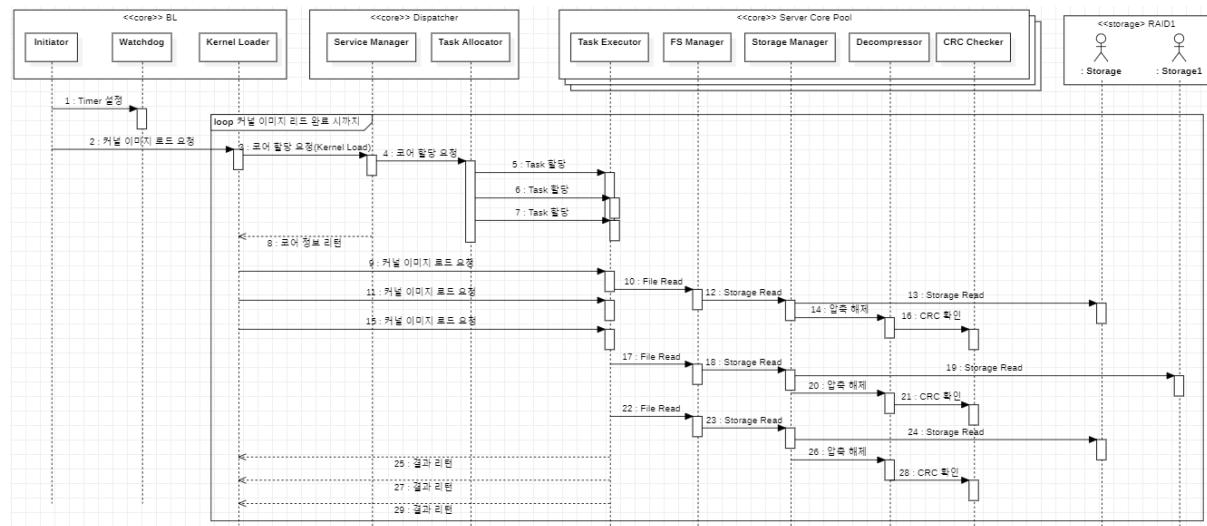


그림 10. Kernel Image 로드 Sequence Diagram.

위 그림은 Initiator가 커널 Image 로드 수행을 위한 Sequence Diagram을 나타낸다. Kernel Image 로드는 Multi-Core를 사용한 병렬화로 성능을 향상 시킨다. 따라서 예상치 못한 Hang을 대비하기 위해, Initiator는 Watchdog에게 Timer 설정을 요청한다. 그후, Kernel Loader에게 커널 이미지 요청을 하며, Kernel Loader는 Dispatcher Core의 Service Manager에게 코어를 할당 받는다. 이 때, Task Allocator는 Kernel Loader가 요청한 Task에 대해 Unit Operation (Ex. 4KB Read)으로 나누고, 수행에 필요한 수만큼의 Server Core를 Pool에서 할당해준다.

Kernel Loader는 할당 받은 Server Core들에게 각각 커널 이미지 로드를 요청하며, 각 Server Core들은 압축된 커널 이미지를 읽어와, 압축 해제 및 CRC 체크 후, 그 결과를 리턴 해준다. 이 때, RAID1 구성되어 있는 2개의 Storage로부터 Server Core들이 병렬적으로 커널 이미지를 읽어온다.

3.1.3. Kernel 변조 체크

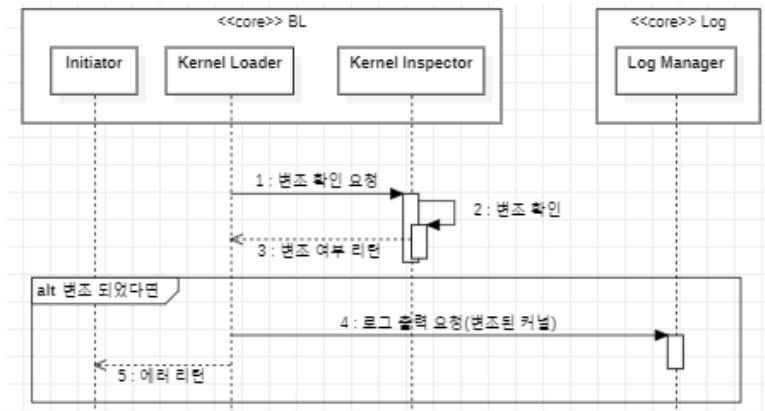


그림 11. Kernel 변조 체크 Sequence Diagram.

위 그림은 Kernel 변조 체크에 대한 Sequence Diagram이다. Kernel Image 로드가 완료되면, Kernel Loader는 Kernel Inspector에게 커널 이미지 변조 확인을 요청한다. Kernel Inspector는 변조 여부를 확인하며, 그 결과를 리턴 한다. 만약, 커널 이미지가 변조되었다면, Kernel Loader는 Log Core의 Log Manager에게 로그 출력을 요청하며, Initiator에게 에러를 리턴 한다.

3.1.4. Shell Mode 진입 대기

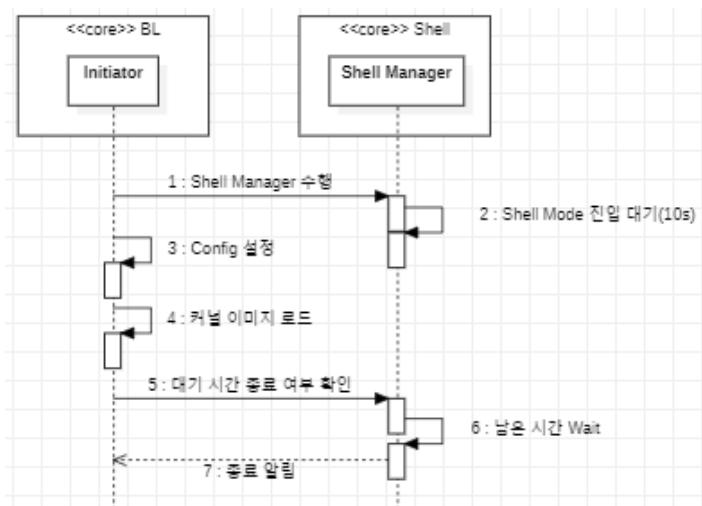


그림 12. Shell Mode 진입 대기 Sequence Diagram.

위 그림은 Shell Mode 진입을 위한 대기 시간을 확인하기 위한 Sequence Diagram이다. Initiator는 Kernel 수행 전, 사용자가 Shell Mode 진입키를 누르기 위해 특정 시간 (10s) 대기하여야 한다. 따라서 해당 시간을 앞당기기 위해, Initiator는 가장 먼저 Shell Manager를 수행한다. Initiator는 모든 부팅 준비가 끝난 시점에 Shell Manager에게 대기 시간이 완료되었는지를 체크한다. 만약 대기 시간이 남아 있다면, Shell Manager는 해당 시간을 Wait한 후, 종료 후, Initiator에게 알려준다.

3.1.5. Kernel 수행

Initiator는 부팅에 필요한 모든 동작을 마무리하고, Kernel에게 제어권을 넘긴다. 해당 동작은 Initiator가 수행 중인 BL Core에서 진행된다.

3.2. UC_02에 대한 동작 상세

아래 그림은 UC_02에 대한 시스템 관점에서의 Sequence Diagram이다. 본 장에서는 이를 바탕으로 각 단계에 대한 Core별 상세 동작 구조를 기술한다.

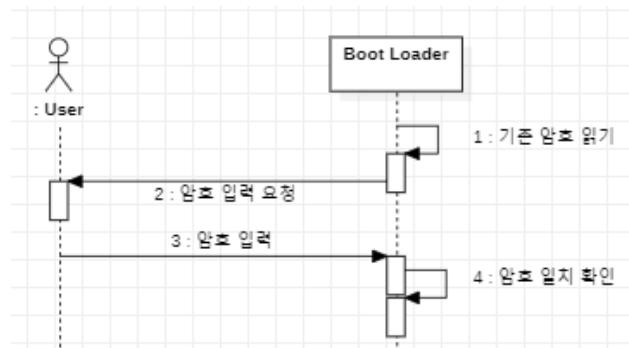


그림 13. 시스템 관점의 Password 입력 Sequence Diagram.

3.2.1. 기존 암호 읽기

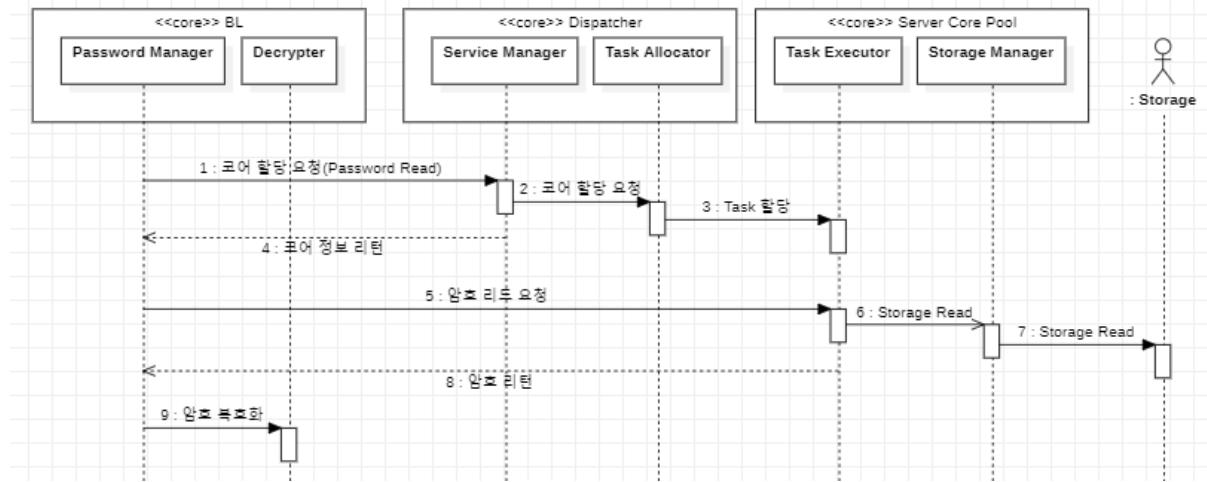


그림 14. 기존 암호 읽기 Sequence Diagram.

위 그림은 Password Manager가 Storage 장치에 저장된 기존 암호를 읽어오는 Sequence Diagram이다. Password Manager는 Storage 접근을 위해 Dispatcher Core에게 코어 할당을 요청한다. 이후, 할당 받은 Server Core에 암호 리드를 요청하며, 암호 리드가 끝나면, 이를 복호화한다.

3.2.2. 사용자 암호 입력 및 일치 확인

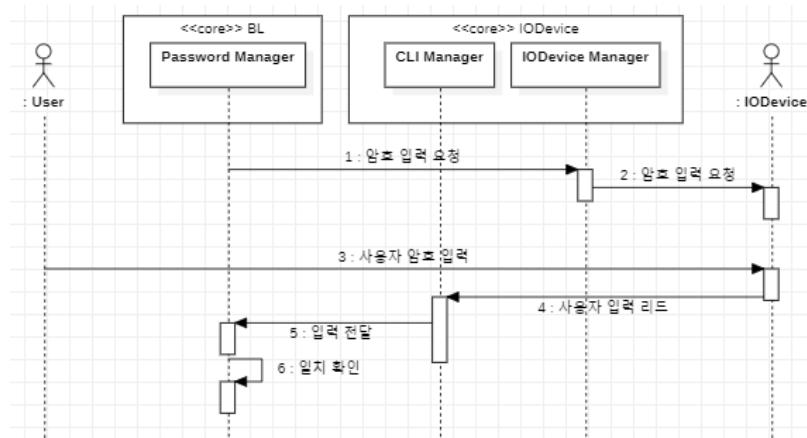


그림 15. 사용자 암호 입력 Sequence Diagram.

위 그림은 Password Manager가 사용자에게 암호 입력을 요청하고, 사용자가 입력한 암호를 IODevice Manager로부터 받아, 일치 여부를 확인하는 Sequence Diagram이다. IODevice Core는

모니터를 통해, 사용자에게 암호 입력 요청을 하고, 키보드를 통해 사용자가 입력한 암호를 리드한다.

3.3. UC_03에 대한 동작 상세

아래 그림은 UC_03에 대한 시스템 관점에서의 Sequence Diagram이다. 본 장에서는 이를 바탕으로 각 단계에 대한 Core별 상세 동작 구조를 기술한다.

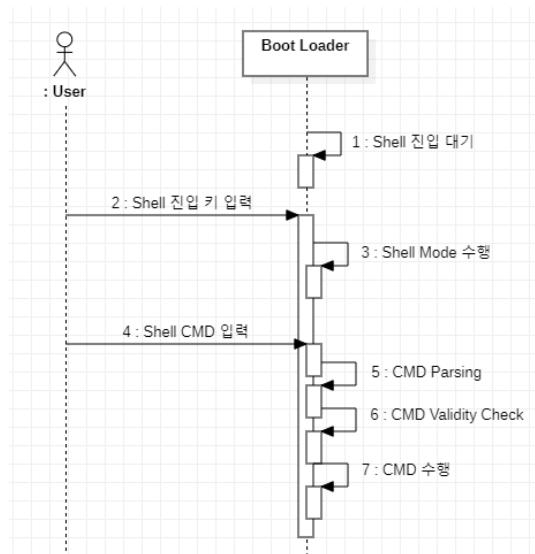


그림 16. 시스템 관점의 Shell Command 수행 Sequence Diagram.

3.3.1. Shell Mode 진입 및 커맨드 입력

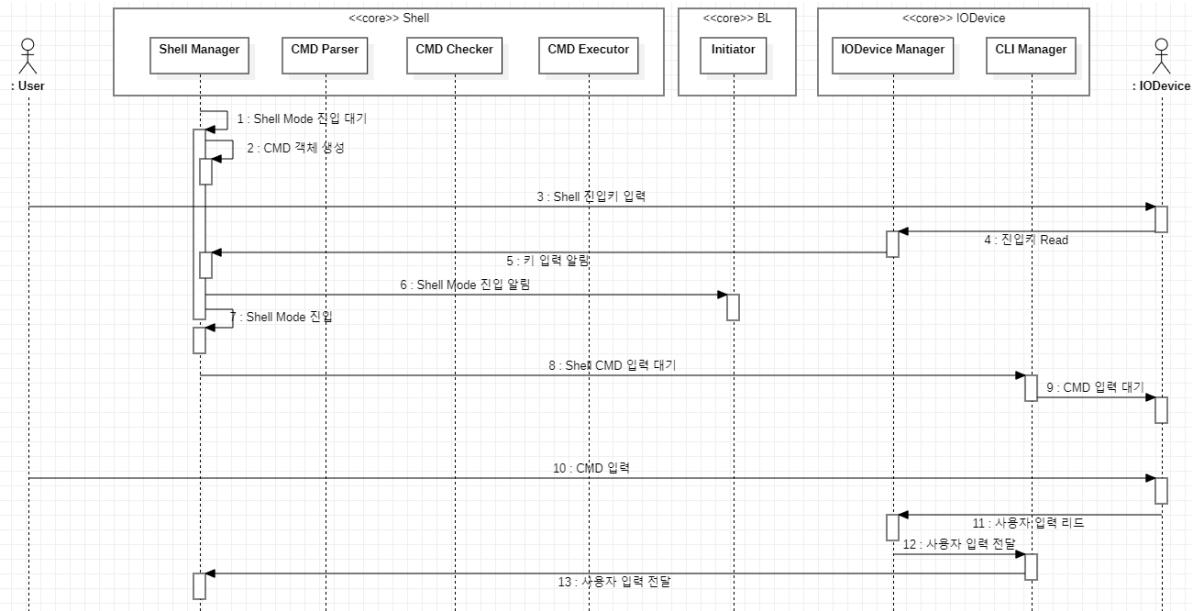


그림 17. Shell Mode 수행 Sequence Diagram

위 그림은 Shell Manager가 Shell Mode 진입 대기 과정에서 사용자가 진입키를 입력하여, Shell Mode로 진입하는 Sequence Diagram이다. Shell Manager는 Boot Loader 시작 시점에 수행되며, Shell Mode 진입을 대기 중이며, 이 과정에서 사용자가 키보드를 통해 Shell 진입키를 입력하면, 이를 IODevice Core로부터 전달받아, Shell Mode로 진입한다. 그 후, 사용자가 입력한 Shell Command는 IODevice Manager, CLI Manager를 거쳐 Shell Manager로 전달된다.

3.3.2. Shell Command 수행

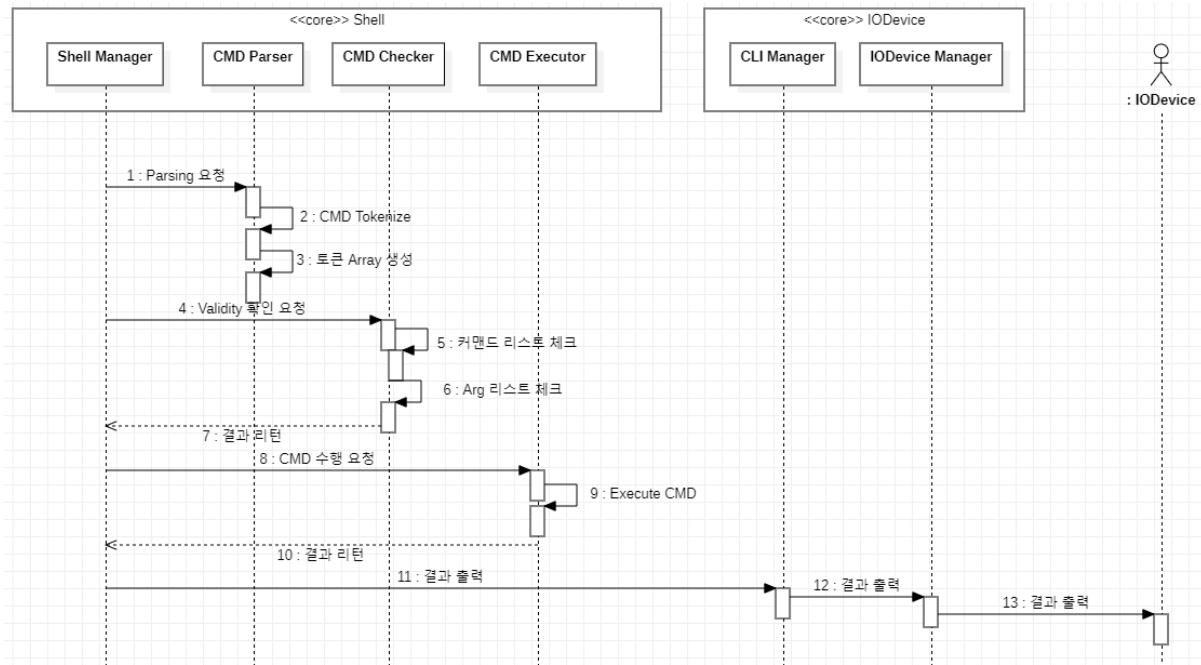


그림 18. Shell Command 수행 Sequence Diagram

위 그림은 Shell Manager가 입력 받은 사용자 커맨드를 수행하는 Sequence Diagram이다. Shell Manager는 CMD Parser에게 Tokenize 및 토큰 Array 생성을 요청하며, 그 결과를 바탕으로 CMD Checker에게 유효성 확인을 요청한다. 그 후, CMD Executor를 통해, 커맨드를 수행하고, IODevice를 통해 결과를 제공한다.

4. 모듈 사양

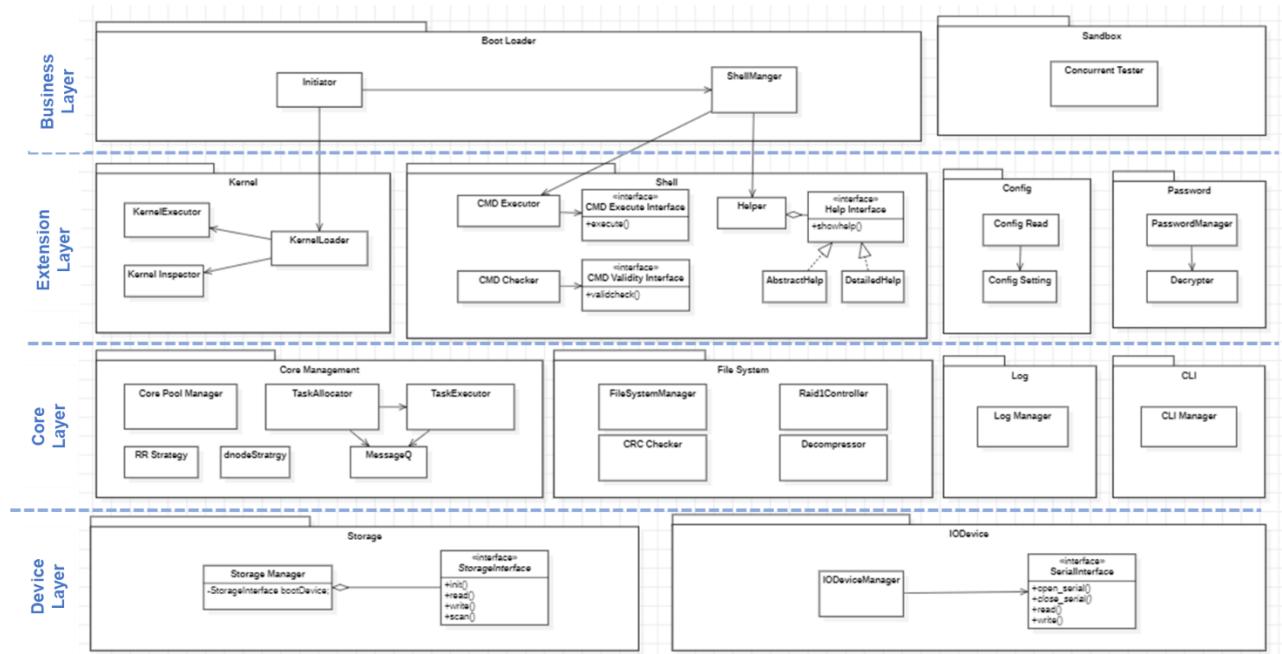


그림 14. 개발 측면에서의 Module View.

위 그림은 본 과제에서 제안하는 Boot Loader의 Module View이다. Layered Style로 구성되어 있으며, 4개의 Layer가 있다. 각 Layer에 대한 설명은 다음과 같다.

- **Device Layer:** 가장 하단 위치의 Layer로 경계인 Storage와 IO Device 접근에 필요한 인터페이스를 제공한다.
- **Core Layer:** Device Layer에 속하는 모듈들을 직접적으로 참조하는 모듈들이 속하는 Layer로, 상위 Layer가 필요한 병렬 처리, 파일 시스템 접근, 로그 출력, 사용자 커맨드 처리 관련 기능을 제공하는 모듈들이 위치한다.
- **Extension Layer:** Core Layer 모듈들을 제공하는 기능과 인터페이스를 통해, 확장된 기능을 제공하는 Layer로, Kernel 로딩 및 Kernel 수행을 담당하거나, Shell Command에 대한 부문 동작들을 수행하는 모듈들이 위치한다.
- **Business Layer:** 전체 Business Logic을 제공하는 최상위 Layer

4.1. Device Layer

Device Layer는 Storage와 IO Device Package로 구성된다.

4.1.1. Storage Package

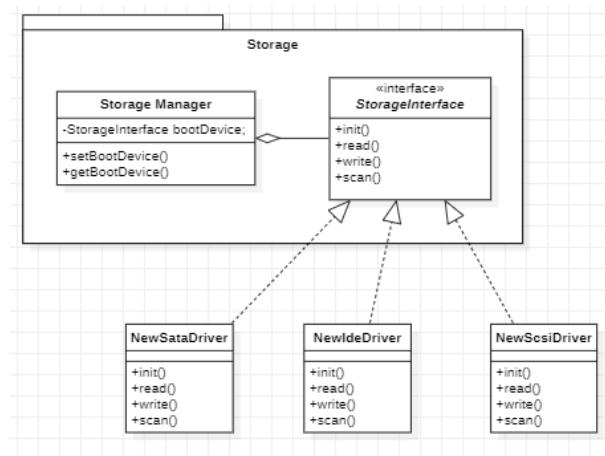


그림 20. Storage Package.

Storage 장치 관리 및 접근을 위한 모듈들이 모여 있다. Storage Interface는 Storage 장치 추가 및 변경이 용이하기 위해, De-factor Standard 인터페이스를 채택하였다. 제조사가 제공하는 스토리지 Driver는 이를 상속받아 구현되어 있다. 추가로, 사용자가 Shell Command를 통해 다른 Storage 장치 선택을 지원하기 위해, 서버에 장착된 Storage 장치에 대한 객체들을 관리한다. 해당 객체들은 Storage Device Scan 시점에 생성되며, 빠른 탐색을 위해 Hash Table로 관리된다.

4.1.2. IODevice Package

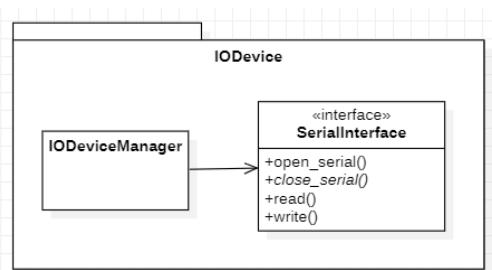


그림 21. IODevice Package.

키보드, 모니터 등 직렬 포트를 통한 사용자 입출력 장치를 위한 모듈들이 모여 있다. Serial Interface는 IODevice 추가 및 변경이 용이하기 위해, De-factor Standard 인터페이스를 채택하였으며, 제조사가 제공하는 IODevice Driver는 이를 상속받아 구현되어 있다.

4.2. Core Layer

Core Layer는 Core Management, File System, Log, CLI Package로 구성된다.

4.2.1. Core Management Package

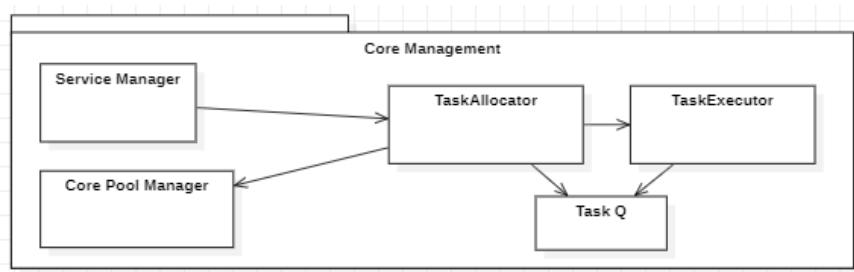


그림 22. Core Management Package.

Multi-Core를 활용한 병렬처리 기능을 위한 모듈들이 위치한다.

- Core Pool Manager는 다수의 Server Core들은 Pool을 관리한다. Dedicated Core를 제외한 나머지 모든 Core에 대한 정보를 가지고 있다.
- Service Manager는 Client로부터 요청이 오면, Task Allocator에게 실제 Task 할당을 의뢰하고, 할당된 Core 정보를 Client에게 전송한다.
- Task Allocator는 다수의 Server Core에게 Task를 배정하는 역할을 수행한다. 이 때, Server Core 간의 균등한 Task 배분을 위해, 기 정의된 Unit Operation (Filesystem Traverse: dnode 기준, Storage Read: 4KB)으로 Task를 나눈다. Server Core에게 Task Queue를 통해, Task를 배정하며, Round Robin 순서로 배정한다.

4.2.2. File System Package

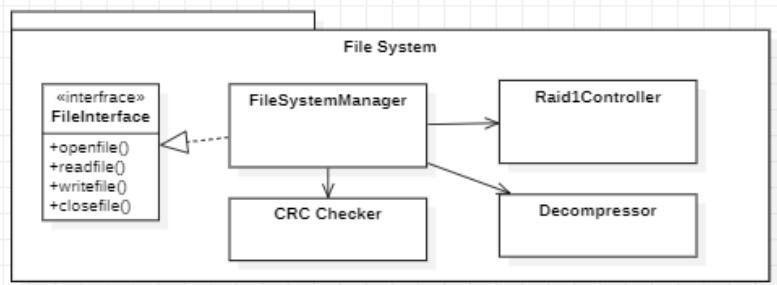


그림 23. File System Package.

상위 Layer에게 File Interface를 통해, File 접근 기능을 제공하는 모듈들이 위치한다. File Read 시, RAID1으로 구성된 2개의 스토리지에서 병렬적으로 데이터를 읽어온다. 추가로, 압축된 커널 이미지의 경우, 압축 해제를 수행하며, CRC 확인 결과 이상이 없는 파일 리드 결과만 전달한다.

4.2.3. Log Package

IODevice를 통해, Log 출력 기능을 담당하는 패키지이다. Log는 각 모듈별로 이미 정의되어 있으며, Log Manager는 Log ID를 통해, Log 출력을 요청 받는다.

4.2.4. CLI Package

IODevice를 통해 키보드로부터 사용자 입력을 받아, 이를 문자열 형태로 전달하는 동작이 구현되는 패키지이다.

4.3. Extension Layer

Extension Layer는 Kernel과 Shell, Config, Password Package로 구성된다.

4.3.1. Kernel Package

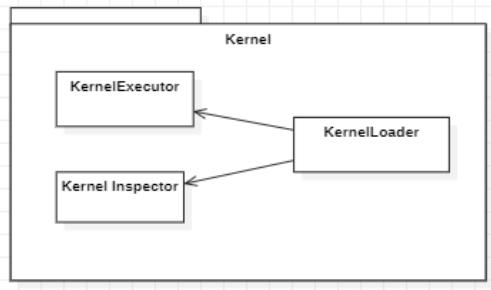


그림 24. Kernel Package.

커널 로딩을 담당하는 모듈들이 모여 있는 패키지이다. Kernel Loader는 커널 로딩 및 커널에게 제어권을 넘겨주기 위한 로직을 담당하는 모듈이다. Config Manager가 설정한 위치의 커널 이미지를 DRAM으로 읽어 온 후, Kernel Inspector를 통해 Kernel 변조 여부 확인하고, 이상이 없을 경우, Kernel을 수행한다.

4.3.2. Shell Package

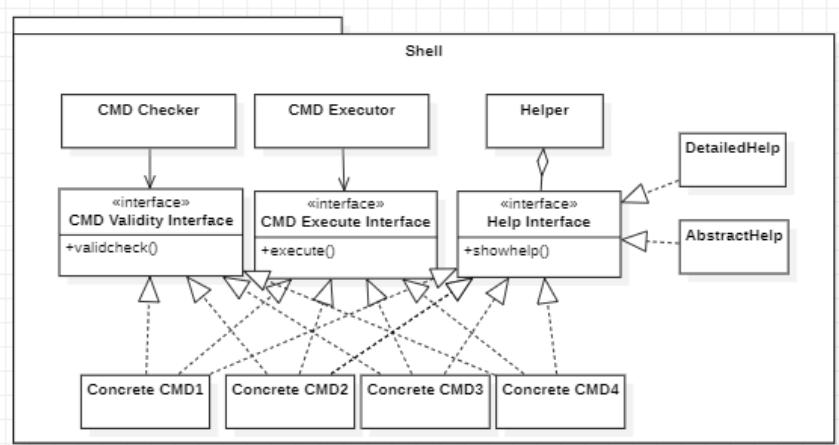


그림 25. Shell Package.

Shell Command 수행을 위해 필요한 부문 동작들을 구현된 패키지이다. 커맨드 수행을 위해 공통으로 필요한 인터페이스는 총 3가지로 각각에 대해 ISP를 적용하여 각각의 모듈로 분리하였다. Concrete Command Class는 총 3가지의 인터페이스를 다중 상속받아 구현하여야 한다. Helper 모듈의 경우, 사용자의 입력에 따라 간략 도움말과 상세 도움말 제공을 위해 전략 패턴을 적용하였

다.

4.3.3. Config Package

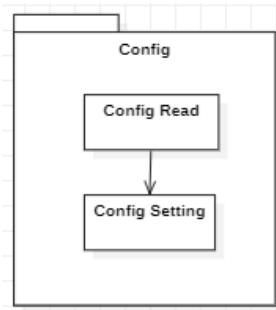


그림 26. Config Package.

Config 설정을 위한 동작들이 구현된 패키지이다. 변경이 빈번한 Config 항목에 대한 부분을 Config Setting 모듈로 분리하였다. Config Read 모듈은 Config 파일로부터 값을 읽어와 Config Setting에 전달한다.

4.3.4. Password Package

사용자 암호 입력에 관련된 동작들이 구현된 패키지이다. Boot Loader 수행 과정에서 사용자 암호가 설정되어 있으면, 사용자로부터 암호를 입력 받아, 해당 암호가 일치하는지 여부를 확인한다. 이 때, 기존 암호는 Encryption된 상태로 저장되어 있으며, 따라서 스토리지 장치로부터 읽어온 암호는 Decryption하여야 한다.

4.4. Business Layer

Business Layer는 Boot Loader Package가 존재한다.

4.4.1. Boot Loader Package

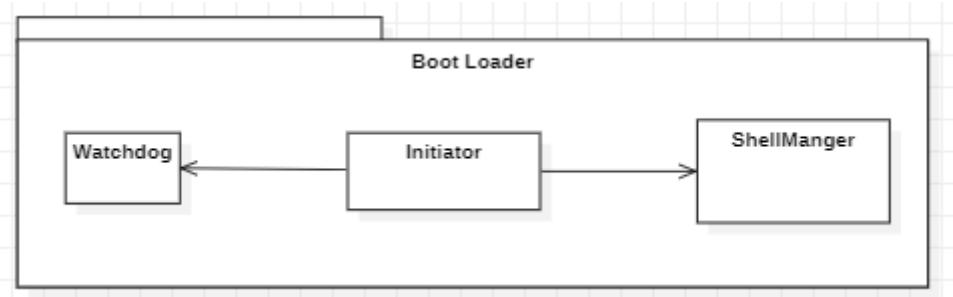


그림 27. Boot Loader Package.

- Initiator: Boot Loader 서비스 위해 필요한 Sequence를 명세하고, Extension Layer에 속하는 모듈을 호출하는 모듈. 예상치 못한 Hang 발생에 대한 빠른 Detect을 위해, 병렬 Core 처리 전 Watchdog Timer를 세팅한다.
- Shell Manager: Shell Mode 제공을 위해 필요한 Sequence를 명세하고 있는 모듈.

부록

A. 도메인 모델.....	43
A1. 도메인 모델.....	43
A2. Sequence Diagram	43
A2.1. UC_01 Boot Loader 수행.....	44
A2.2. UC_02 패스워드 입력	44
A2.3. UC_03 Shell 수행	45
A2.4. UC_04 Boot 환경 설정	46
B. 품질 시나리오	47
C. 품질 시나리오 분석	50
C1. 품질 시나리오 별 중요도 및 복잡도 평가 근거.....	50
C2. 품질 속성 선정 근거.....	52
D. 후보 구조.....	54
D1. QA_01 Boot Loader 수행 시간	54
D1.1. CA1-1 Client-Server Core 도입	56
D1.2. CA1-2 Dispatcher Core 추가 도입	57
D1.3. CA1-3 코어 간 Shared Memory를 통한 통신	59
D1.4. CA1-4 코어 간 Task Queue를 통한 통신	59
D1.5. CA1-5 Client Core와 비동기 (Asynchronous) 수행 가능 컴포넌트를 Server Core에	



(주) 보이는 소프트웨어 연구소
조용진(drajin.cho@bosornd.com)

할당	60
D1.6. CA1-6 단일 컴포넌트를 다수의 Server Core에 할당	61
D1.7. CA1-7 Server Core Pool 관리	62
D1.8. CA1-8 Round Robin 할당	63
D1.9. CA1-9 남은 Task가 가장 적은 Core 할당.....	63
D1.10. CA1-10 Sandbox Testing 도입	63
D1.11. CA1-11 병렬 처리 Trace 모듈 추가	64
D1.12. CA1-12 Server Core간 inode 기준 탐색 병렬화	64
D1.13. CA1-13 Server Core간 dnode 기준 탐색 병렬화.....	65
D1.14. CA1-14 커널 이미지 파일 inode 위치 별도 저장	66
D1.15. CA1-15 커널 이미지 압축 저장.....	66
D1.16. CA1-16 커널 이미지 병렬 압축 해제	67
D1.17. CA1-17 커널 이미지 이중화 (RAID 1).....	67
D1.18. CA1-18 Shell Manager 최우선 수행	68
D2. QA_02 디바이스 추가 및 변경 용이성.....	69
D2.1. CA2-1 Storage Device 추상화를 위한 De-factor Standard 인터페이스 사용	70
D2.2. CA2-2 스토리지 타입 별 Adapter 적용	72
D2.3. CA2-3 제조사별 추가 추상화	73
D2.4. CA2-4 DIP 원칙 적용	74
D2.5. CA2-5 Storage Device 객체 관리 클래스 정의.....	75
D2.6. CA2-6 Scan 시점에 미리 객체 생성하기	76

D2.7. CA2-7 Linked List를 통한 Device 객체 관리	76
D2.8. CA2-8 Hash Table을 통한 Device 객체 관리	77
D2.9. CA2-9 IO Device 추상화를 위한 De-factor Standard 인터페이스 사용	77
D2.10. CA2-10 Device Manager 세분화.....	78
D2.11. CA2-11 컴포넌트 세분화	79
D2.12. CA2-12 고수준 Layer 설계	80
D3. QA_03 Shell Command 처리 시간	82
D3.1. CA3-1 순차 Parsing.....	84
D3.2. CA3-2 Array를 통한 토큰 관리.....	85
D3.3. CA3-3 커맨드 및 Argument 리스트 저장을 위한 Linked List 사용	85
D3.4. CA3-4 커맨드 및 Argument 저장을 위한 Hash Table 사용	86
D3.5. CA3-5 커맨드 및 Argument 저장을 위한 Trie 사용	86
D3.6. CA3-6 파일 시스템 Traverse 병렬화	87
D3.7. CA3-7 커널 이미지 로드 병렬화.....	88
D3.8. CA3-8 커맨드 객체 미리 생성.....	88
D3.9. CA3-9 디렉토리 파일 Bloom Filter 사용	89
D3.10. CA-10 Cache 도입	90
D4. QA_04 커널 이미지 로딩 가능성	91
D4.1. CA4-1 커널 이미지 파일 Header CRC 관리	91
D4.2. CA4-2 커널 이미지 부분적 CRC 관리	92
D4.3. CA4-3 CRC 체크 병렬화.....	92

D4.4. CA4-4 Watchdog Timer 활용.....	92
D4.5. CA4-5 CRC 에러 발생 시 Reload.....	93
D4.6. CA4-6 ECC 활용 복구.....	93
D4.7. CA4-7 ECC 메모리 채택.....	93
D4.8. CA4-8 Fail 감지 시점에 알리기	93
D5. QA_05 Shell Command 추가 및 확장 용이성.....	95
D5.1. CA5-1 Shell Command 필요 IF 도출.....	96
D5.2. CA5-2 Command Pattern 적용	96
D5.3. CA5-3 ISP (Interface Segregation Principle) 적용	97
D5.4. CA5-4 DIP (Dependency Inversion Principle) 적용.....	98
D5.5. CA5-5 Help CMD 전용 모듈 정의	98
D5.6. CA5-6 전략 패턴 정의.....	99
E. 후보 구조 평가.....	100
E1. QA_01 Boot Loader 수행 시간	102
E1.1. CA1-1 Client/Server Core 도입 [채택].....	102
E1.2. CA1-2 Dispatcher Core 추가 도입 [채택].....	102
E1.3. CA1-3 코어 간 Shared Memory를 통한 통신 [미채택]	103
E1.4. CA1-4 코어 간 Task Queue를 통한 통신 [채택]	103
E1.5. CA1-5 Client Core와 비동기 수행 컴포넌트를 Server Core에 할당 [채택]	104
E1.6. CA1-6 단일 컴포넌트를 다수의 Server Core에 할당 [채택].....	104
E1.7. CA1-7 Server Core Pool 관리 [채택].....	104

E1.8. CA1-8 Round Robin 할당 [채택].....	104
E1.9. CA1-9 남은 Task가 가장 적은 Core 할당 [미채택].....	104
E1.10. CA1-10 Sandbox Testing 도입 [채택].....	105
E1.11. CA1-11 병렬처리 Trace 모듈 추가 [미채택]	105
E1.12. CA1-12 Server Core간 inode 기준 탐색 병렬화 [미채택]	106
E1.13. CA1-13 Server Core간 dnode 기준 탐색 병렬화 [채택]	106
E1.14. CA1-14 커널 이미지 파일 inode 위치 별도 저장 [채택].....	106
E1.15. CA1-15 커널 이미지 압축 저장 [채택]	106
E1.16. CA1-16 커널 이미지 병렬 압축 해제 [채택].....	106
E1.17. CA1-17 커널 이미지 이중화 (RAID 1) [채택]	107
E1.18. CA1-18 Shell Manager 최우선 수행 [채택]	107
E2. QA_02 디바이스 추가 및 변경 용이성.....	108
E2.1. CA2-1 Storage Device를 위해 De-factor Standard 인터페이스 사용 [채택].....	108
E2.2. CA2-2 스토리지 타입 별 Adapter 적용 [채택].....	108
E2.3. CA2-3 제조사별 추가 추상화 [미채택]	108
E2.4. CA2-4 DIP 원칙 적용 [채택]	108
E2.5. CA2-5 Storage Device 객체 관리 클래스 정의 [채택]	109
E2.6. CA2-6 Scan 시점에 미리 Storage Device 객체 생성하기 [채택]	109
E2.7. CA2-7 Linked List를 통한 Device 객체 관리 [미채택]	110
E2.8. CA2-8 Hash Table을 통한 Device 객체 관리 [채택]	110
E2.9. CA2-9 IO Device를 위해 De-factor Standard 인터페이스 사용 [채택].....	110

E2.10. CA2-10 Device Manager 세분화 [채택]	111
E2.11. CA2-11 컴포넌트 세분화 [채택]	111
E2.12. CA2-12 고수준 Layer 설계 [채택]	111
E3. QA_03 Shell Command 처리 시간	112
E3.1. CA3-1 커맨드 순차 Parsing [채택]	112
E3.2. CA3-2 Array를 통한 토큰 관리 [채택]	113
E3.3. CA3-3 커맨드/Arg 리스트 관리를 위한 Linked List 사용 [미채택]	113
E3.4. CA3-4 커맨드/Arg 리스트 관리를 위한 Hash Table 사용 [채택]	113
E3.5. CA3-5 커맨드/Arg 리스트 관리를 위한 Trie 사용 [미채택]	113
E3.6. CA3-6 파일 시스템 Traverse 병렬화 [채택]	113
E3.7. CA3-7 커널 이미지 로드 병렬화 [채택]	113
E3.8. CA3-8 커맨드 객체 미리 생성 [채택]	114
E3.9. CA3-9 디렉토리 파일 Bloom Filter 구축 [미채택]	114
E3.10. CA3-10 Cache 추가 [미채택]	115
E4. QA_04 커널 이미지 로딩 가용성	116
E4.1. CA4-1 커널 이미지 Header CRC 관리 [채택]	116
E4.2. CA4-2 커널 이미지 부분적 CRC 관리 [채택]	116
E4.3. CA4-3 CRC 체크 병렬화 관리 [채택]	116
E4.4. CA4-4 Watchdog Timer 활용 [채택]	117
E4.5. CA4-5 CRC 에러 발생 시 Reload [미채택]	117
E4.6. CA4-6 ECC 활용 커널 이미지 복구 [미채택]	118

E4.7. CA4-7 ECC 메모리 채택 [채택].....	118
E4.8. CA4-8 Fail 감지 시점에 알리기 [채택].....	118
E5. QA_05 Shell Command 추가 및 확장 용이성	118
E5.1. CA5-1 Shell Command 필요 인터페이스 도출 [채택].....	119
E5.2. CA5-2 Command Pattern 적용 [채택].....	119
E5.3. CA5-3 Command 인터페이스에 ISP 적용 [채택]	120
E5.4. CA5-4 DIP 적용 [채택].....	120
E5.5. Help CMD 전용 모듈 정의 [채택].....	120
E5.6. 도움말 출력 전략 패턴 적용 [채택]	120
F. 최종 구조 설계	121
F1. 시스템 동작/실행 최종 구조 설계 경위.....	121
F1.1. CPU 관련 최종 구조	122
F1.2. DRAM 관련 최종 구조	123
F1.3. Storage 관련 최종 구조.....	123
F1.4. IO Device 관련 최종 구조.....	123
F2. 개발 측면 최종 구조 설계 경위.....	124
F2.1. Device Layer	125
F2.2. Core Layer	125
F2.3. Extension Layer	126
F2.4. Business Layer	126
F3. 최종 구조의 단점 및 Risk.....	127

A. 도메인 모델

A1. 도메인 모델

본 문서에서 설계하는 Boot Loader의 도메인 모델은 아래 그림과 같다. 해당 도메인 모델은 Boundary-Control-Entity 패턴을 이용하여 Use Case를 기반으로 Sequence Diagram을 작성할 때 도출되는 컴포넌트로부터 도출되었다. Entity는 시스템 데이터를 나타내는 컴포넌트이며, Boundary는 시스템 액터와 인터페이스 하는 컴포넌트이다. Control은 Boundary와 Entity 사이를 연결해주며, 명령을 수행한다. 일반적으로 Use Case 수행을 위한 비즈니스 로직을 담고 있다.

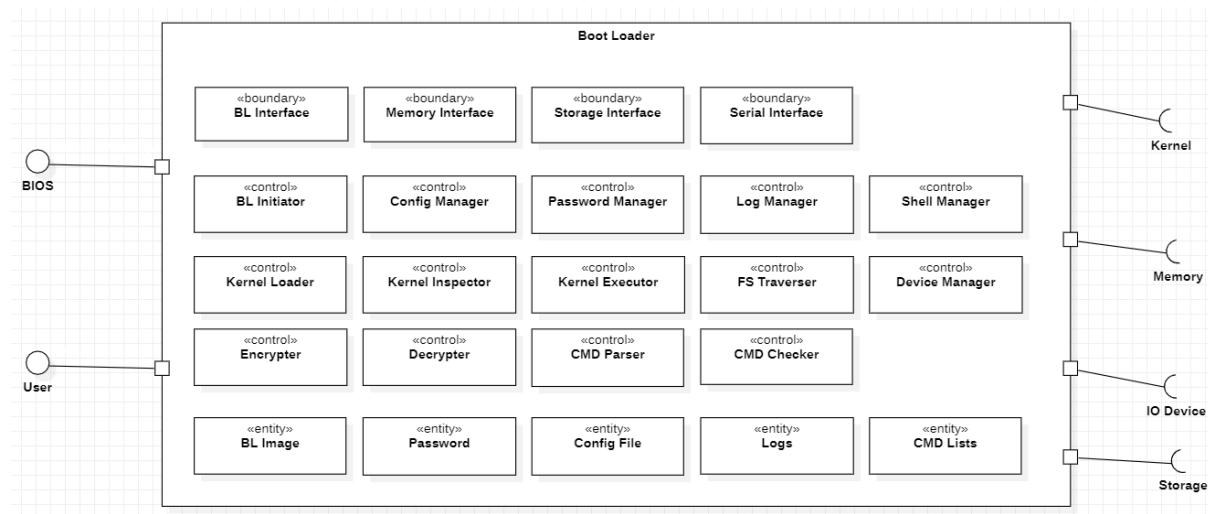


그림 28. 도메인 모델.

A2. Sequence Diagram

각 Use Case에 대한 세부 Sequence Diagram은 다음과 같다.

A2.1. UC_01 Boot Loader 수행

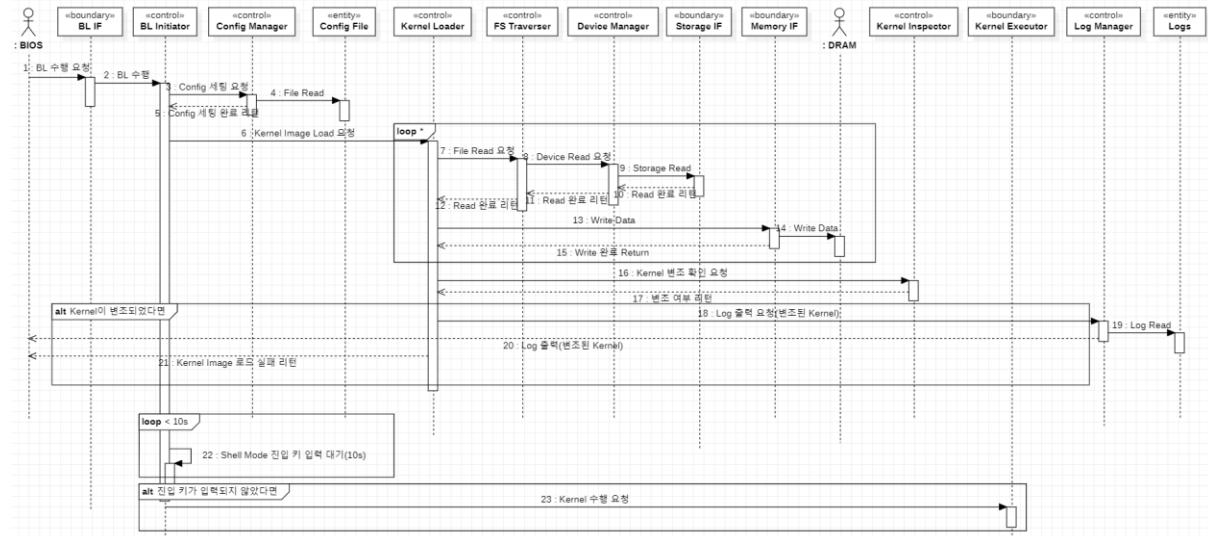


그림 29. UC_01 Sequence Diagram (Kernel Image 로드 및 수행).

A2.2. UC_02 패스워드 입력

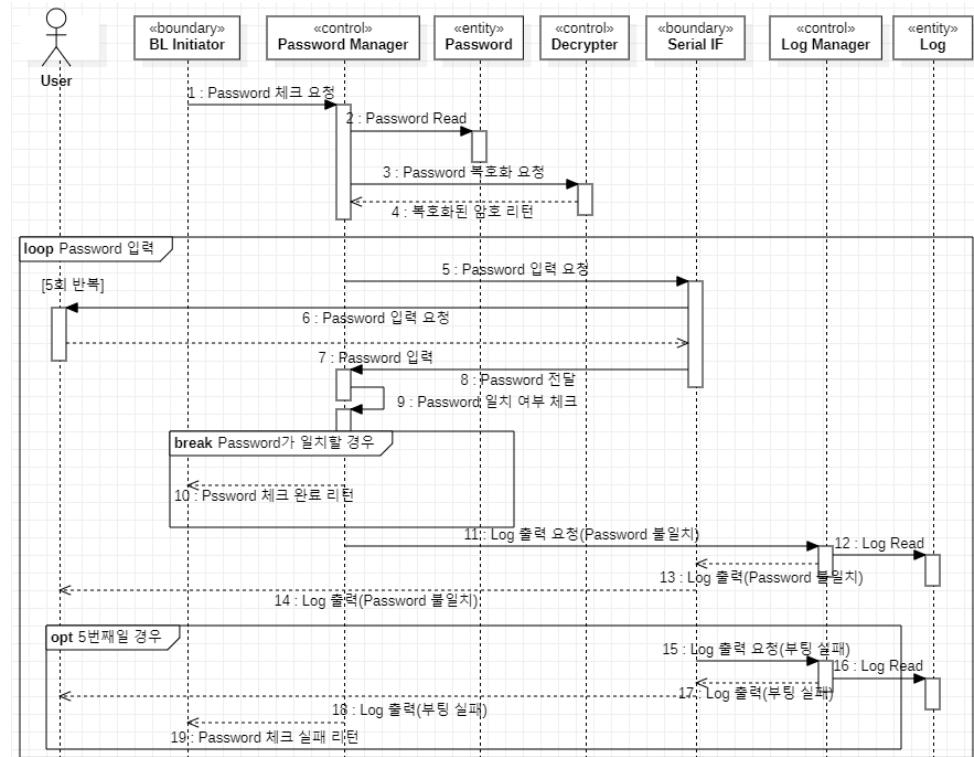


그림 30. UC_02 Sequence Diagram (패스워드 입력).

A2.3. UC_03 Shell 수행

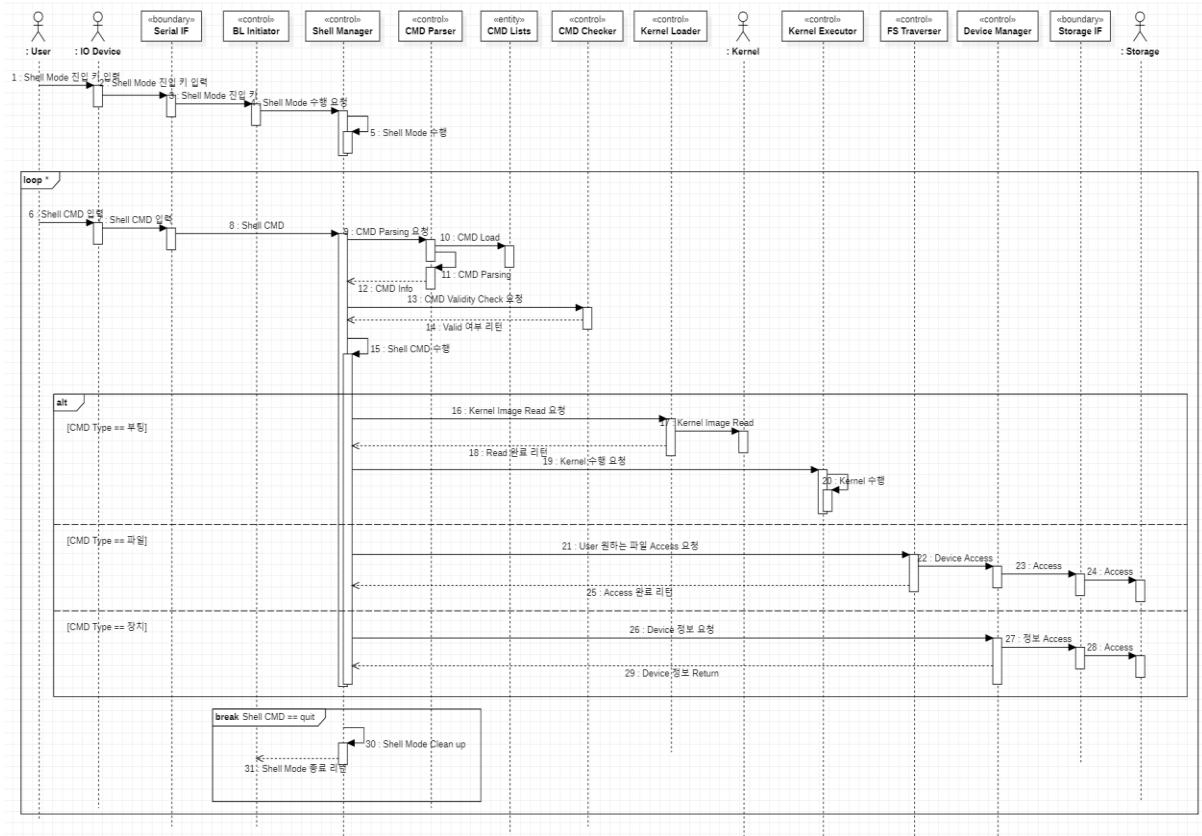


그림 31. UC_03 Sequence Diagram (Shell 수행).

A2.4. UC_04 Boot 환경 설정

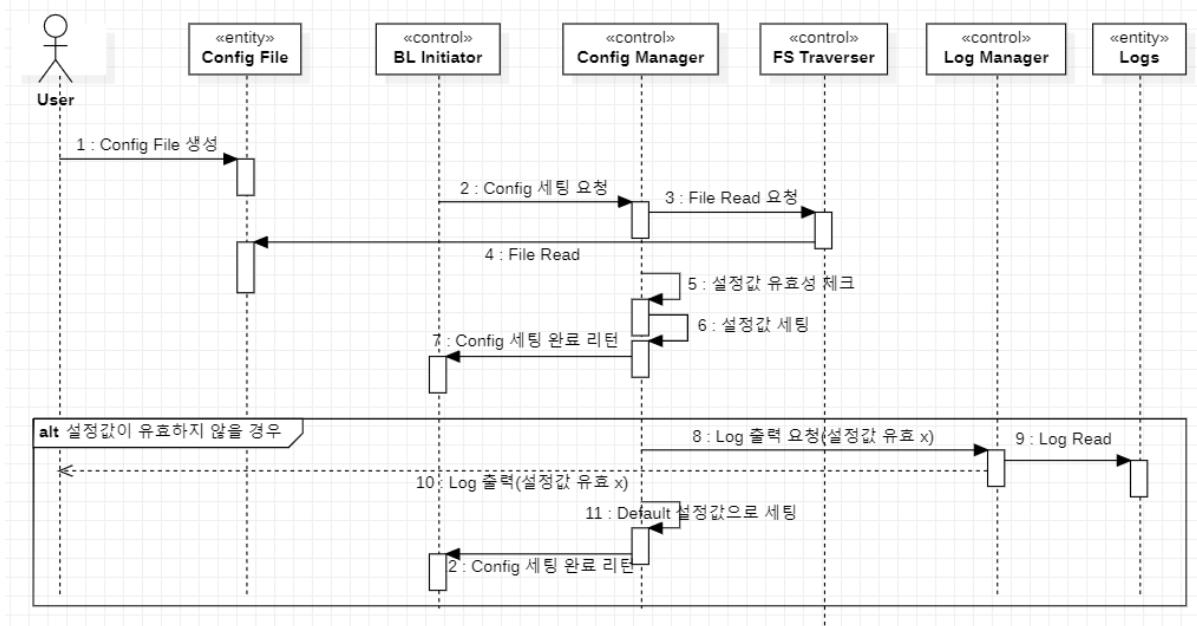


그림 32. UC_04 Sequence Diagram (Boot 환경 설정).

B. 품질 시나리오

품질 시나리오 도출을 위해, Stakeholder 중 사용자와 개발자 관점에서의 Role Play를 진행하였다. Role Play를 통해 아래와 같은 품질 요구사항을 도출하였다.

- [사용자 (서버 운영자)]

- 서버 부팅 시간이 빨랐으면 좋겠다.
- 부팅이 실패하는 일이 없었으면 좋겠다.
- Shell Command에 대한 응답이 빨리 오면 좋겠다.
- 다양한 Shell Command를 지원했으면 좋겠다.
- 일부 사용자의 기본적인 공격은 자동으로 막아줬으면 좋겠다.

- [개발자]

- 제품 경쟁력을 위해 Boot Loader 동작 시간을 짧게 하고 싶다.
- 부팅 실패 시, 어떤 문제인지 빨리 Detect하고 싶다.
- Device가 변경 혹은 추가될 때 작업량이 적었으면 좋겠다.
- Shell Command 변경 혹은 추가될 때 작업량이 적었으면 좋겠다.
- 새로운 패치 적용이 빠르면 좋겠다.

위의 품질 요구사항을 바탕으로 도출한 품질 시나리오는 아래와 같다.

QS_01	Performance	Boot Loader 수행 시간
시나리오	Boot Loader는 Kernel 부팅에 필요한 모든 동작을 10s 내에 반드시 완료하여야 한다. 추가로 부팅 시간은 짧을수록 좋다. [부팅 시간] = [Kernel 수행 시작] – [Boot Loader 수행 시작 시작]	
QS_02	Performance	Shell Mode 진입 시간

시나리오	사용자가 Shell Mode 진입 키를 입력하면, Boot Loader에서 Shell Mode로의 전환 시간은 짧을수록 좋다 [Shell Mode 진입 시간] = [Shell Mode에서 사용자 입력 가능 시각] – [Shell Mode 진입 키 입력 시각]	
QS_03	Performance	Shell Command 처리 시간
시나리오	Boot Loader는 사용자가 입력한 Shell Command 빠르게 처리하여야 한다. Shell Command 수행 시간은 짧을수록 좋다 [Shell Command 수행 시간] = [Shell Command 입력 시각] – [Shell Command 결과 리턴 시각]	
QS_04	Performance	신규 패치 업데이트 속도
시나리오	Boot Loader는 새로운 패치를 Update하는데 시간이 짧을수록 좋다. [Update 시간] = [Update 완료 시각] – [Update 시작 시각]	
QS_05	Availability	Boot Loader 손상 복원 시간
시나리오	Boot Loader는 Self-Test 중 발견한 이미지 손상 복원 시, 해당 복원 시간은 짧을수록 좋다. [손상 복원 시간] = [손상 복원 알고리즘 종료 시각] – [손상 복원 알고리즘 시작 시각]	
QS_06	Availability	커널 이미지 로딩 가능성
시나리오	Boot Loader 수행 과정에서 장애 발생 시, 정상 상태로 빨리 돌아가야 한다. [정상 복구 시간] = [Recover 완료 시각] – [Fail 발생 시각]	
QS_07	Availability	빠른 장애 발생 보고
	Boot Loader 수행 과정에서 장애 발생 시, 최대한 빨리 사용자에게 알려야 한다. 장애 보고에 걸린 시간은 짧을수록 좋다. [장애 보고에 걸린 시간] = [사용자 보고 시각] - [실제 장애 발생 시각]	
QS_08	Security	Kernel Image 변조 감지
시나리오	사용자가 승인되지 않는 Kernel 이미지로 부팅 시도 시, Boot Loader는 변조된 Kernel의 수행을 막아야 한다. 변조 커널 부팅 성률은 낮을수록 좋다. [변조 커널 부팅 성공률] = [부팅 성공 횟수/ [Kernel Image 변조 후 부팅 시도 횟수]]	
QS_09	Modifiability	디바이스 추가 및 변경 용이성
시나리오	Storage, IO Device의 다양한 디바이스가 추가되거나, 이미 장착된 디바이스 드라이버가 변경되었을 때, 시스템 변경을 위한 개발 비용이 최소화되어야 한다. [개발 비용 (M/M)] = [수정, 검증을 다시 해야 하는 모듈] / [파일의 크기(LoC)]	

QS_10	Modifiability	Shell Command 추가 및 변경 용이성
시나리오		Shell Command에 대한 추가 및 변경 개발 비용이 최소화되어야 한다. [개발 비용 (M/M)] = [수정, 검증을 다시 해야 하는 모듈] / [파일의 크기(LoC)]
QS_11	Modifiability	Config 추가 및 변경 용이성
시나리오		신규 Configuration이나 기존 Configuration 변경 시 개발 비용이 최소화되어야 한다. [개발 비용 (M/M)] = [수정, 검증을 다시 해야 하는 모듈] / [파일의 크기(LoC)]

C. 품질 시나리오 분석

Stakeholder들의 Role Play로부터 도출된 비즈니스 드라이버는 다음과 같다.

1. Boot Loader는 최대한 빨리 부팅 작업을 완료하여, 사용자의 만족도를 높여야 한다.
2. 개발할 Boot Loader는 하드웨어 장치 추가 및 변경에 대해 신속하게 대응하여 원하는 기간 내에 제품 개발을 완료할 수 있어야 한다.
3. Booting 과정에서 발생하는 장애 상황으로 인한 사용자의 불편은 최소화되어야 한다.
4. 일부 사용자의 악의적인 공격에 대한 대비가 충분히 되어있어야 한다.

위의 비즈니스 드라이버를 바탕으로 품질 시나리오 중요도 및 복잡도에 대한 분석과 비기능적 요구사항/품질 속성 선정 결과는 다음과 같다.

품질 시나리오		중요도	복잡도	선정 결과
Performance	QS_01 Boot Loader 수행 시간	H	H	NFR_01 QA_01
	QS_02 Shell Mode 진입 시간	H	L	
	QS_03 Shell Command 처리 시간	H	H	QA_03
	QS_04 신규 패치 업데이트 속도	L	M	
Availability	QS_05 Boot Loader 손상 복원 시간	L	M	
	QS_06 커널 이미지 로딩 가용성	H	M	QA_04
	QS_07 빠른 장애 발생 보고	H	L	
Security	QS_08 Kernel Image 변조 감지	L	M	
Modifiability	QS_09 디바이스 추가 및 변경 용이성	H	H	QA_02
	QS_10 Shell Command 추가 및 변경 용이성	M	M	QA_05
	QS_11 Config 추가 및 변경 용이성	H	L	

C1. 품질 시나리오 별 중요도 및 복잡도 평가 근거

QS_01 Boot Loader 수행 시간은 Boot Loader가 갖추어야 할 가장 기본적인 성능 품질로, 시스템 사용자가 체감하는 가장 중요한 품질 속성이다. 따라서 중요도를 H로 선정하였다. 수행 시간 최적화를 위해서는 Boot Loader 수행 시작부터 종료까지 모든 과정이 최적화되어야 하며, 현재 서

버 시스템 기준으로 10s 내에 부팅 시간은 업계 최고 수준의 부팅 시간이다. 따라서 복잡도를 H로 평가하였다.

QS_02 Shell Mode 진입 시간은 일반적으로 데이터센터 서버에서 장애 발생이나 중요 평가 시나리오와 관련이 있다. 따라서, 중요도는 H로 평가하였다. 다만, 서버 Boot Loader의 경우, 추가적인 POST 과정이 필요하지 않아, 시작 시점에 Shell Mode 진입이 바로 가능하다. 따라서 복잡도는 L로 평가하였다.

QS_03 Shell Command 처리 시간은 빠른 장애 해결로 이어지게 되며, 중요도를 H로 평가하였다. 구현에 있어서는 Shell Command 시작부터 사용자에게 결과 리턴까지 모든 과정이 최적화되어야 하며, 다양한 Shell Command에 대한 각각의 최적화 기술이 필요하다. 따라서 복잡도는 H로 평가하였다.

QS_04 신규 패치 업데이트 속도의 중요도는 L로 평가하였다. 일반적으로 데이터센터는 수백 대 ~ 수천 대 서버를 일괄적으로 업데이트하는 경우가 빈번하며, 이 경우 Boot Loader에서 제공해주는 기능보다는 전용 툴을 사용하는 경우가 많다. 복잡도의 경우, 패치 시간 동안 서비스를 진행하지 않아도 되기 때문에 복잡도는 M으로 평가하였다.

QS_05 Boot Loader 손상 복원 시간은 Boot Loader 이미지가 저장된 MBR에 장애가 발생했을 때 나타나는 시나리오로, 데이터센터에서는 매우 빈번히 발생하는 시나리오이다. 다만, 부팅 스토리지는 RAID1을 구성하는 것이 일반적이라 중요도를 L로 평가 (RAID 복구 가능)하였다. 손상 복원의 경우, 널리 알려진 ECC 기법을 수행하는 것이 좋다고 판단하여 복잡도는 M으로 평가하였다.

QS_06 커널 이미지 로딩 가용성은 장애 발생 시, 얼마나 빨리 정상 상태로 돌아가느냐에 대한 가용성 관련 품질 속성으로, 안정적인 데이터센터 서비스 운영을 위해 중요하다. 따라서 중요도를 H로 하였다. 장애 복구를 위한 방법들은 다양하며, Watchdog Timer 등에 대한 개발이 필요하여, 개발 난이도는 M로 평가하였다.

QS_07 빠른 장애 발생 보고는 역시도 중요한 가용성 관련 품질 속성으로 중요도를 H로 평가하였다. 다만, 보고를 위한 Alert 기능은 개발 난이도가 낮아, L으로 평가하였다.

QS_08 Kernel Image 변조 감지는 보안 품질 속성이다. 다만, 가상화된 데이터센터 서버의 특성상, 외부 사용자에 의한 Kernel Image 변조 시도는 구조적으로 어렵다. 따라서 중요도는 L로 하였으며, 오픈소스로 공개된 변조 탐지 알고리즘을 채용할 시 중요도를 M으로 평가하였다.

QS_09 디바이스 추가 및 변경 용이성은 시스템 전체적인 구조를 결정하는 품질 속성이자, 다양 Boot Loader

한 디바이스 장착이 필요한 데이터센터 서버에 반드시 필요한 품질 시나리오로 중요도를 H로 평가하였다. 추가로, 설계 의존적인 부분이 많아, 복잡도를 H로 하였다.

QS_10 Shell Command 추가 및 변경 용이성은 디바이스 추가 및 변경 용이성과 비교하여 상대적으로 발생 빈도가 적은 품질 시나리오로 중요도를 M로 평가하였다. 또한, 개발에 있어, Kernel 구현 방식을 참고할 수 있어, 복잡도를 M으로 평가하였다.

QS_11 Configuration 변경 편리성은 개발 과정에서 많이 발생하는 품질 시나리오로 중요도는 H로 평가하였다. 다만, 파일 방식을 통해 쉽게 달성을 가능하므로, 복잡도는 L로 평가하였다.

C2. 품질 속성 선정 근거

품질 시나리오		중요도	복잡도	선정 결과
Performance	QS_01 Boot Loader 수행 시간	H	H	QA_01
	QS_03 Shell Command 처리 시간	H	H	QA_03
Availability	QS_06 커널 이미지 로딩 가용성	H	M	QA_04
Modifiability	QS_09 디바이스 추가 및 변경 용이성	H	H	QA_02
	QS_10 Shell Command 추가 및 변경 용이성	M	M	QA_05

본 장에서는 위에서 평가한 품질 시나리오 별 중요도 및 복잡도 결과를 바탕으로, 중요도/복잡도가 'H'나 'H'인 품질 시나리오들 중 품질 속성을 선정한 근거를 기술한다. 참고로 품질 속성은 번호가 빠를수록 중요성이 높다.

- **QA_01:** QS_01 Boot Loader 수행 시간은 비즈니스 드라이버 및 Role Play에서 Stakeholder들에게 공통적으로 나온 요구사항이다. 따라서 가장 중요한 품질 시나리오라고 판단하였다.
- **QA_02:** QS_10 디바이스 추가 및 변경 용이성은 사업화를 위한 적기 출시를 결정 짓는 주요 요소가 되기에 중요한 품질 속성으로 판단하였다.
- **QA_03:** QS_03 Shell Command 처리 시간은 가용성과 관련된 중요한 품질 속성으로 판단하였다. QS_06 역시도 가용성과 관련된 품질 속성이나, QS_03의 복잡도가 더 높아, 이를 더욱 중요한 품질 속성으로 판단하였다.

- **QA_04:** QA_06 커널 이미지 로딩 가용성은 위에서 언급한 바와 같이, QS_03과 비교하여 순위가 낮다고 판단하였다.
- **QA_05:** QA_10 Shell Command 추가 및 변경 용이성은 후보 품질 시나리오 좋아 중요도 와 복잡도가 가장 낮아 5순위에 위치시켰다.

D. 후보 구조

본 부록에서는 각 품질 요구사항에 대한 설계 이슈와 이를 해결하기 위한 솔루션 (후보 구조)를 도출한다. 본 부록에서는 이해를 돋기 위해 아래와 같은 범례를 사용하였다.



그림 33. 후보 구조 범례.

D1. QA_01 Boot Loader 수행 시간

QA_01의 품질 측정 정의는 아래와 같다.

- [부팅 시간] = [Kernel 수행 시작] – [Boot Loader 수행 시작]

아래 그림은 데이터센터 서버용 Boot Loader 수행에 대한 Sequence Diagram (그림 29)을 나타낸다. 본 후보 구조에서는 'Kernel Image 로드'와 'Kernel Image 변조 여부 체크', 'Shell Mode 진입 대기'의 설계 이슈 해결을 위한 후보 구조에 대해 상세히 기술한다. 참고로, 다른 수행 과정의 경우, 수행 단계가 매우 단순하고 수행 시간이 짧아, 본 후보 구조에서는 다루지 않는다.

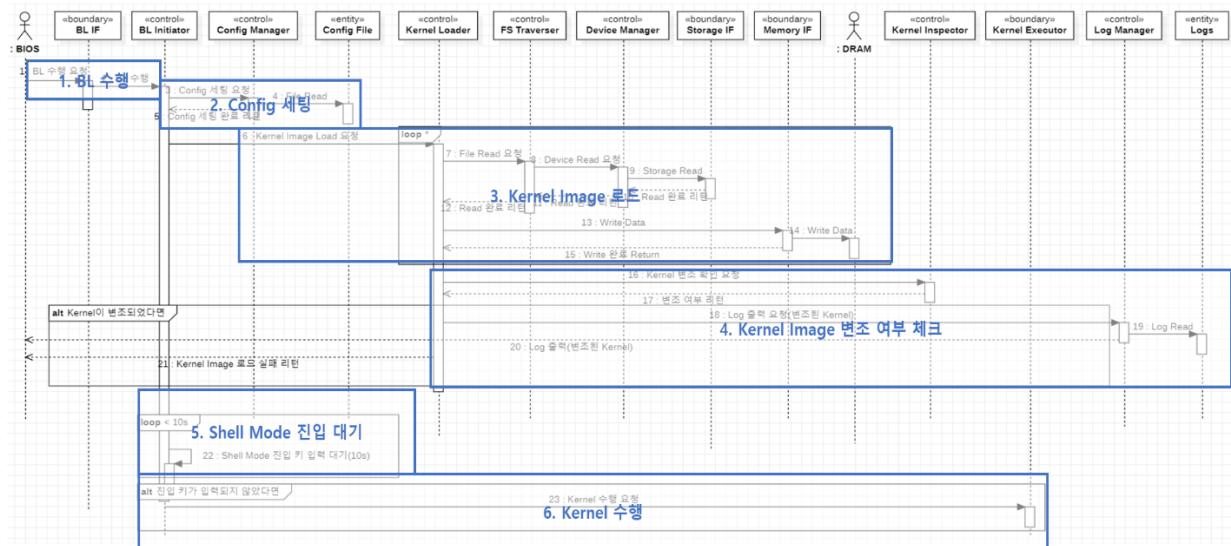
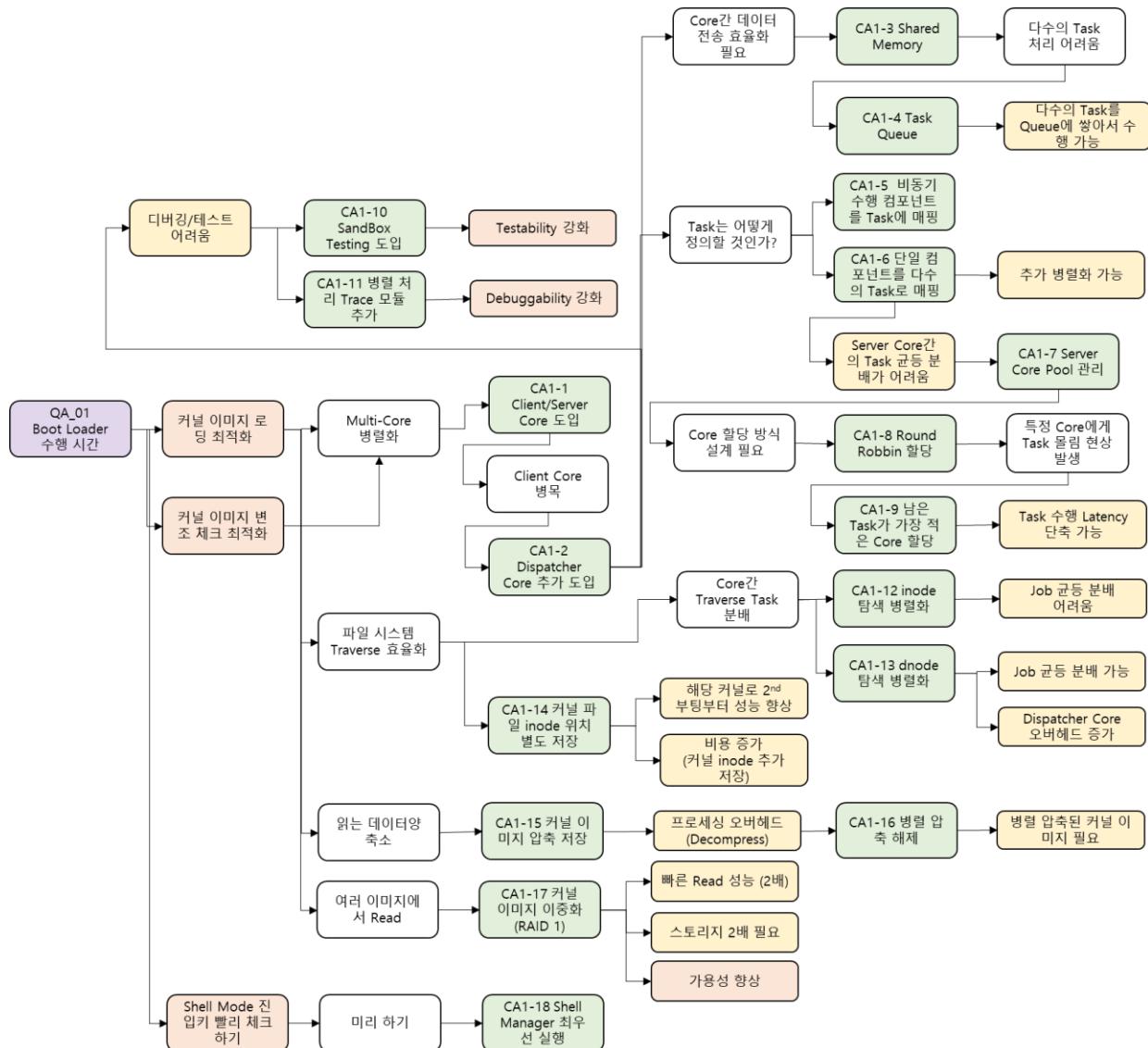


그림 34. 커널 이미지 로드 및 수행 Sequence Diagram.

아래 그림은 QA_01 Boot Loader 수행 시간 관련 후보 구조를 요약한 그림이다.



을 요청하는 방식으로 동작한다.

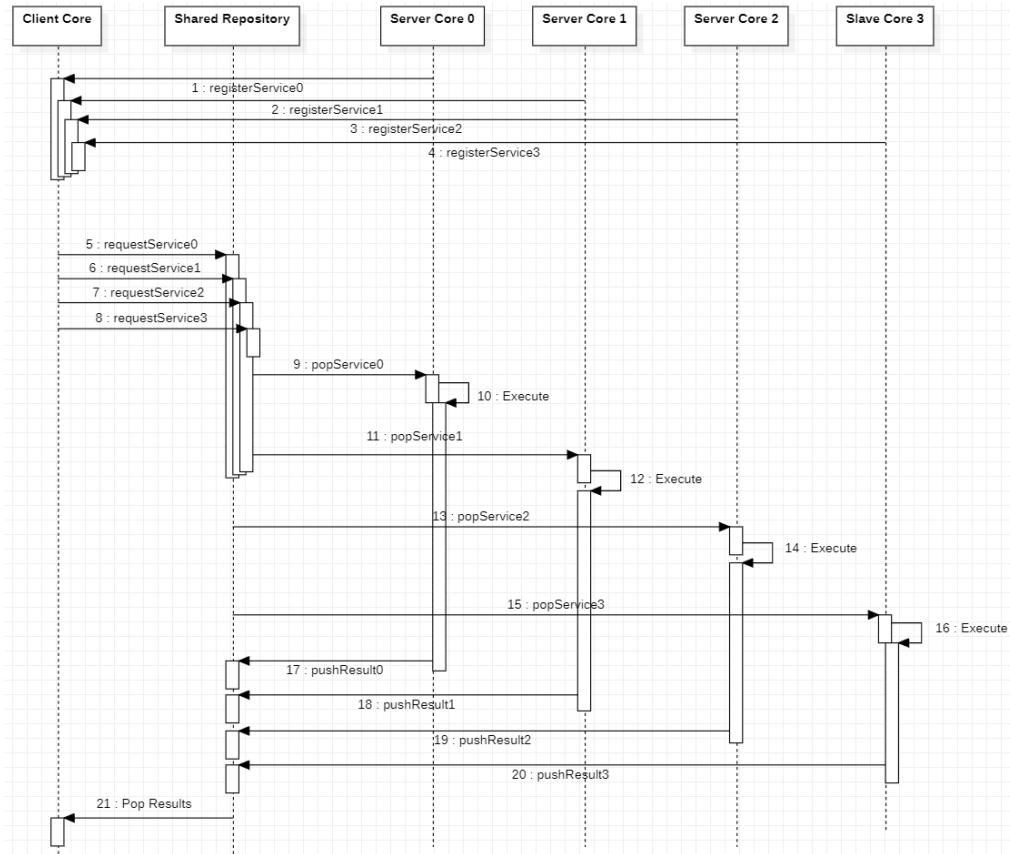


그림 36. Client-Server 구조 Sequence Diagram.

도메인 모델 (그림 28)을 기준으로, Client Core는 Boot Loader의 전체 흐름을 관리하는 BL Initiator 컴포넌트를 수행하고, Server Core들은 나머지 Control 컴포넌트들을 각각 수행한다.

다만, 병렬 처리로 인한 디버깅 및 테스팅 복잡도가 증가하며, 이에 대한 보완 방안 설계가 필요하며, 이는 후보 구조 CA1-10/CA1-11에서 상세히 다룬다.

D1.2. CA1-2 Dispatcher Core 추가 도입

후보 구조 CA1-1은 하나의 Client Core가 BL Initiator 컴포넌트 수행과 더불어 다수의 Server Core에 대한 서비스 관리 (registerService)를 하여야 한다. 따라서 Client Core에 부하가 가중될 수 있으며, 본 후보 구조는 Dispatcher Core 추가 도입하는 방식이다.

아래 그림은 Dispatcher Core가 추가된 구조의 Sequence Diagram을 나타낸다. Dispatcher Core는 (N-2)개의 Server Core가 제공하는 서비스 관리 및 Client Core에서 서비스 수행 요청 시, Server Core와 연결해주는 역할을 수행한다. 그 후, Client Core는 Server Core와 통신을 통해 Task를 병렬 수행한다.

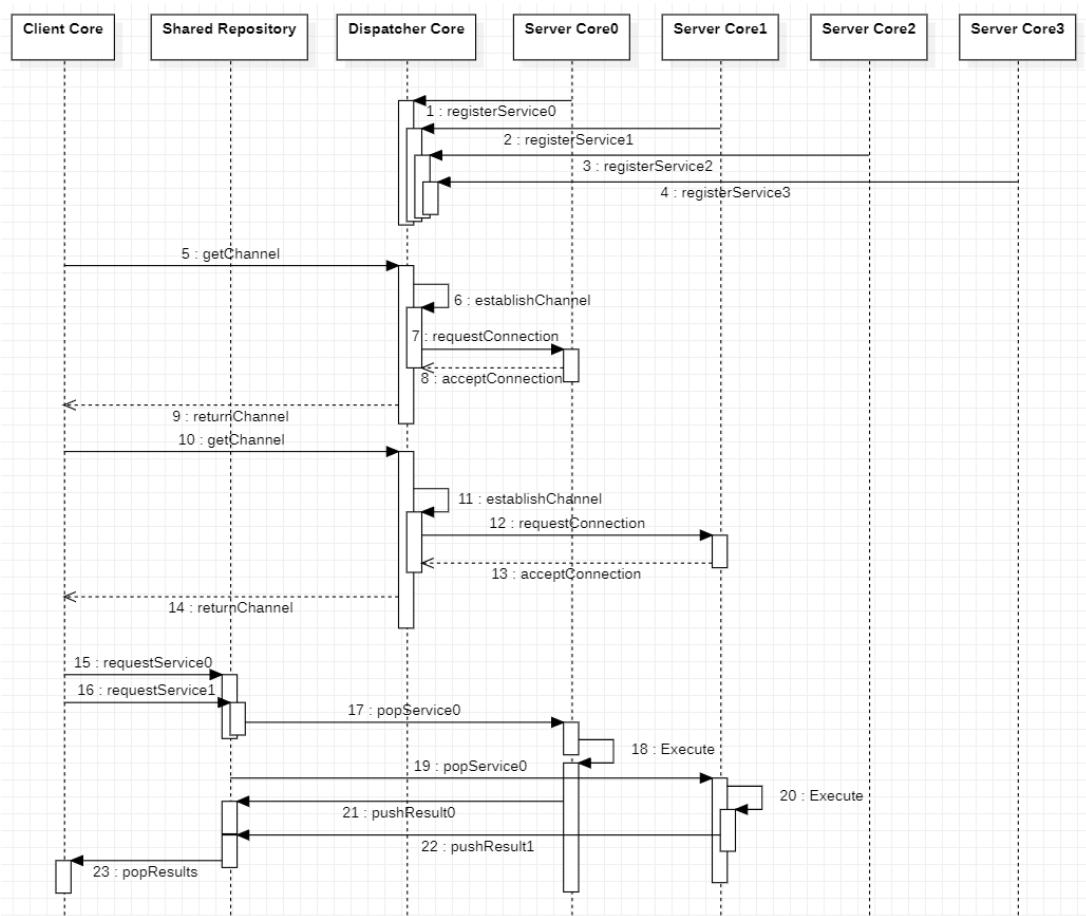


그림 37. Dispatcher Core 추가 구조.

Client-Server 구조는 서비스 요구자 (Client)가 서비스 제공자 (Server)와 직접 통신을 통해, 분산 처리하는 방식이다. 본 시스템에서는 (N-1)개 Server Core가 서로 독립적으로 수행 가능한 Task를 서비스로 등록하고, Client Core가 여러 개의 Server Core에게 해당 Task 작업 수행을 요청하는 방식으로 동작한다.

D1.3. CA1-3 코어 간 Shared Memory를 통한 통신

후보 구조 CA1-1 및 CA1-2에 대한 연장으로, 코어 간 서로 접근 가능한 Shared Memory를 활용하여 Client와 Server가 통신하는 방식이다. 그러나 여러 개의 서비스 요청을 Queuing하기 어려워, Server Core는 하나의 서비스 수행 종료 후, Client Core가 결과를 읽어가고, 다음 서비스를 요청할 때까지 대기해야 하는 딜레이가 발생할 수 있다.

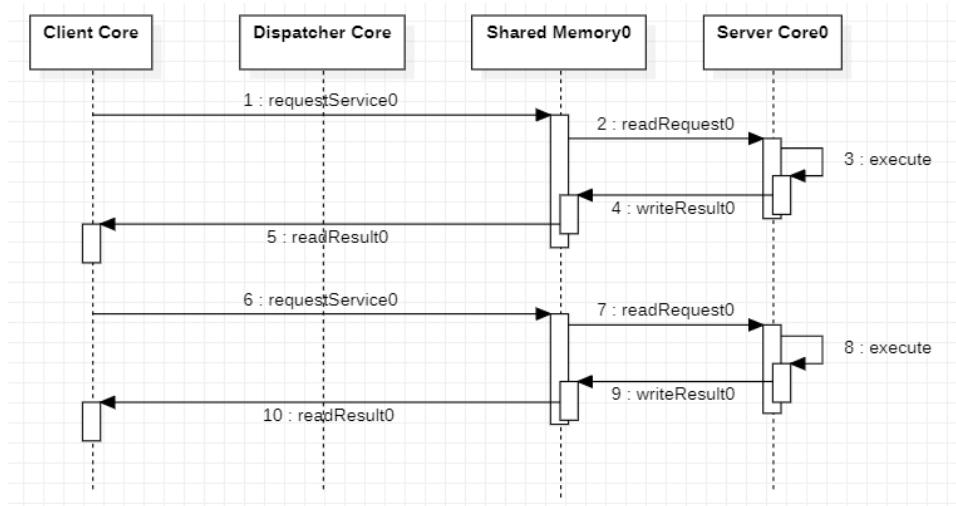


그림 38. Shared Memory 통신 방식.

D1.4. CA1-4 코어 간 Task Queue를 통한 통신

후보 구조 CA1-3의 Risk로 하나의 다수의 Task를 연속적으로 처리하기 어렵다. 따라서 이를 보완하기 위해, 본 후보 구조는 Queue 자료 구조를 도입하는 방식이다. Queue는 선입 선출의 자료구조로, Client Core는 서비스 요청 시, Task Queue에 Push하고, Server Core는 이를 Pop하여 Task를 수행하는 방식이다. Client Core는 다수 개의 서비스 요청을 Queuing할 수 있기 때문에, Server Core에서는 연속적인 Task 수행이 가능하다. Shared Memory 방식과 비교하여 Push/Pop Latency가 발생하지만, 연속적인 Task 수행으로 Server Core에서의 Task 수행 Latency 단축이 가능하다.

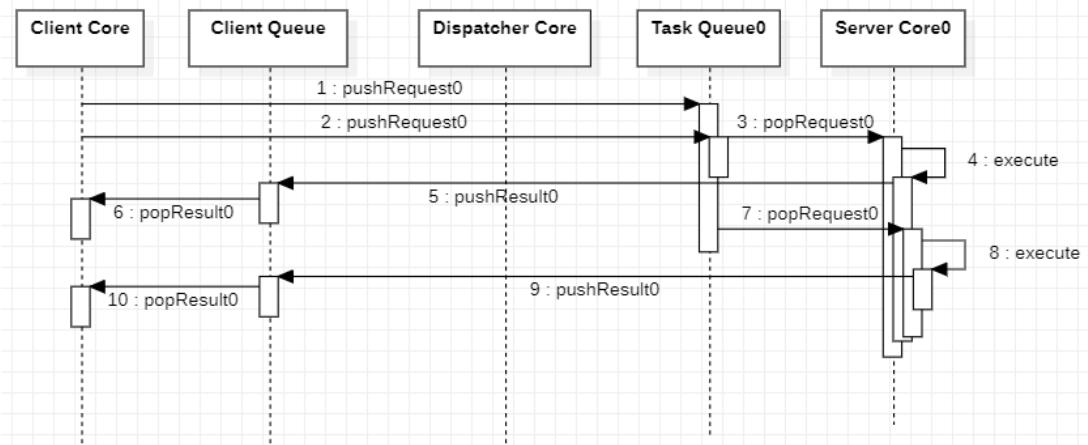


그림 39. Task Queue를 통한 통신 방식.

D1.5. CA1-5 Client Core와 비동기 (Asynchronous) 수행 가능한 컴포넌트를 Server Core에 할당

본 후보 구조에서는 어떠한 Task를 Server Core에서 수행 할지에 대해 구체적으로 다룬다. 만약 어떤 Task를 별도의 Server Core에서 수행하더라도 Client Core가 그 결과를 기다린 후 다음 Task를 수행하여야 한다면, 이로 인한 성능 향상은 없으며, 오히려 Core간의 통신 오버헤드만 추가된다.

따라서 Server Core에서 특정 Task를 수행할 동안, Client Core가 다른 Task를 병렬적으로 수행할 수 있을 경우, 이를 별도의 Server Core에서 수행 가능하다고 판단하였다. 도메인 모델 (그림 28)에서 도출된 Control 컴포넌트 중 비동기 수행 가능한 컴포넌트는 다음과 같다.

- Device Manager: Storage 접근을 담당하는 컴포넌트로, Server Core가 Storage 응답을 기다리는 동안 Client Core는 다른 Task를 수행 (ex. 다음 Storage에 대한 접근 요청)할 수 있다.
- Log Manager: 각종 로그 메시지를 출력해주는 컴포넌트로, Server Core가 상대적으로 느린 IO Device를 통해 로그 메시지를 출력 (ex. 진행 현황에 관한 로그)할 동안, Client Core는 부팅 과정을 계속 진행할 수 있다.
- Shell Manager: 사용자로부터 Shell CMD를 입력 받아 이에 대한 수행하고, 그 결과를 제공하는 컴포넌트. 사용자가 Shell 진입 커맨드를 입력하는지 여부를 별도의 Server Core에서 체크할 수 있다.

아래 그림은 비동기 수행 가능한 컴포넌트를 별도의 Server Core로 할당한 구조이다. Client Core에서는 전체 Flow를 관리하는 BL Initiator 컴포넌트 및 비동기 불가능 컴포넌트들이 수행되며, 총 3개의 Server Core에서 각각 비동기 수행 가능 컴포넌트들을 수행한다.

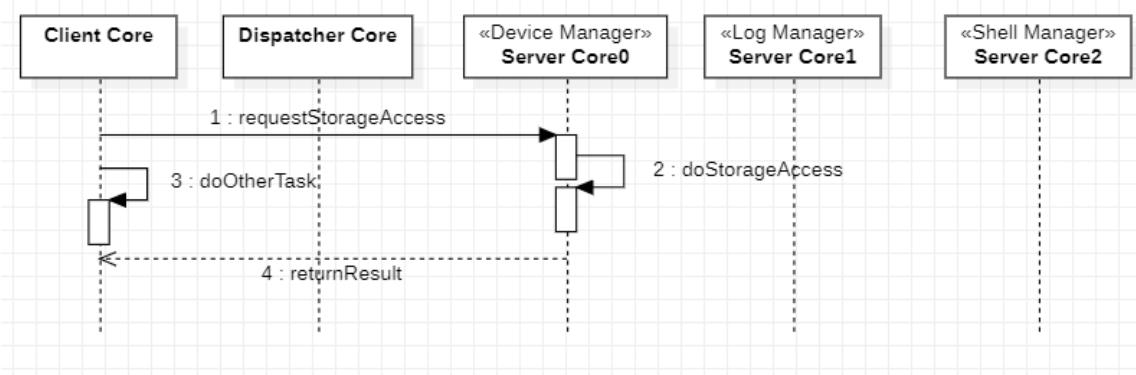


그림 40. 비동기 수행 가능한 컴포넌트들을 별도의 Server Core로 할당.

D1.6. CA1-6 단일 컴포넌트를 다수의 Server Core에 할당

본 후보 구조에서는 동일한 Task를 샤딩 (Sharding)하여 다수의 Server Core에서 수행하는 구조이다. 특히, 수행 시간이 오래 걸리는 Task들은 해당 후보 구조를 통해, 큰 성능 향상을 기대할 수 있다. 아래는 도메인 모델 (그림 28)의 Control 컴포넌트 중 병렬 수행 가능한 컴포넌트들이다.

- Kernel Loader: 데이터센터용 서버를 위한 커널 이미지는 일반적으로 수백 MB 이상이다. 따라서 커널 이미지를 다수의 Server Core를 활용한 스토리지 장치에 병렬 리드 (ex. 커널 이미지가 300MB의 경우, 3개의 Server Core가 0~100MB/100MB~200MB/200MB ~300MB로 오프셋을 나누어 리드)할 경우, 리드 시간을 크게 절약할 수 있다.
- Filesystem Traverser: 커널 이미지 접근을 위해서는 파일 시스템 탐색이 필요하다. 파일 시스템 탐색을 위해서는 inode 접근과 dnode 접근이 반복적으로 발생하며, 이를 다수의 Server Core를 통해 병렬 수행할 경우, 탐색 시간을 크게 단축시킬 수 있다. Filesystem Traverser 컴포넌트의 샤딩 상세 설계는 후보 구조 CA1-12, CA1-3에서 다룬다.
- Kernel Inspector: 읽어온 커널 이미지에 대한 변조 여부를 하나의 코어가 담당하는 것이 아닌 다수의 코어가 이미지 구간을 샤딩하여 체크할 경우, 수행 시간을 단축시킬 수 있다.

병렬 수행 가능한 3개의 컴포넌트 각각은 (N-4)/3개의 Server Core에서 수행된다. 이는 Client Core와 3개의 비동기 컴포넌트 수행 코어를 제외한 나머지 (N-4)개 Server 코어를 3등분한 개수이다.

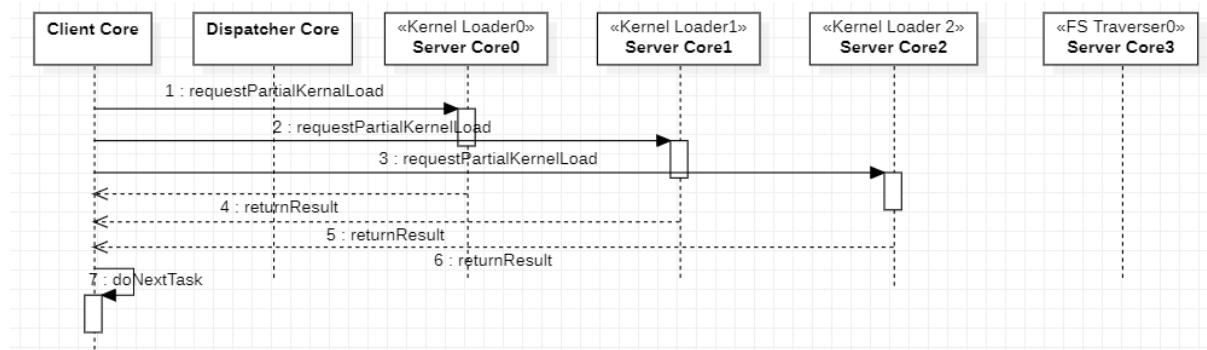


그림 41. 동일 컴포넌트를 색상하여 다수의 Server Core에서 수행.

D1.7. CA1-7 Server Core Pool 관리

후보 구조 C1-6은 각 Server Core가 수행해야 할 Task가 균등하게 분배되지 않는다. 따라서 병렬화 과정에서 Task가 가장 늦게 끝나는 Server Core가 병목이 되는 문제가 있다.

- 컴포넌트 간의 차이: Storage 접근이 필요한 Kernel Loader는 Kernel Inspector보다 수행 시간이 길다.
- 동일 컴포넌트에서 색상 구간에 따른 차이: Filesystem Traverser는 특정 디렉토리의 inode가 크다면 해당 Task를 수행하는 Server Core의 수행 시간이 길다.

또한, Kernel Loader, Filesystem Traverser, Kernel Inspector는 순차적으로 수행되며, 따라서 하나의 컴포넌트가 수행될 때, 나머지 컴포넌트들을 담당하는 Server Core는 유후 상태가 된다.

본 후보 구조에서는 Task를 최소 크기인 Unit Operation으로 색상하고, Server Core들을 Pool로 관리하여, Dispatcher Core가 Client Core에 요청이 있을 때, Pool에 있는 Server Core를 할당해주는 방식이다.

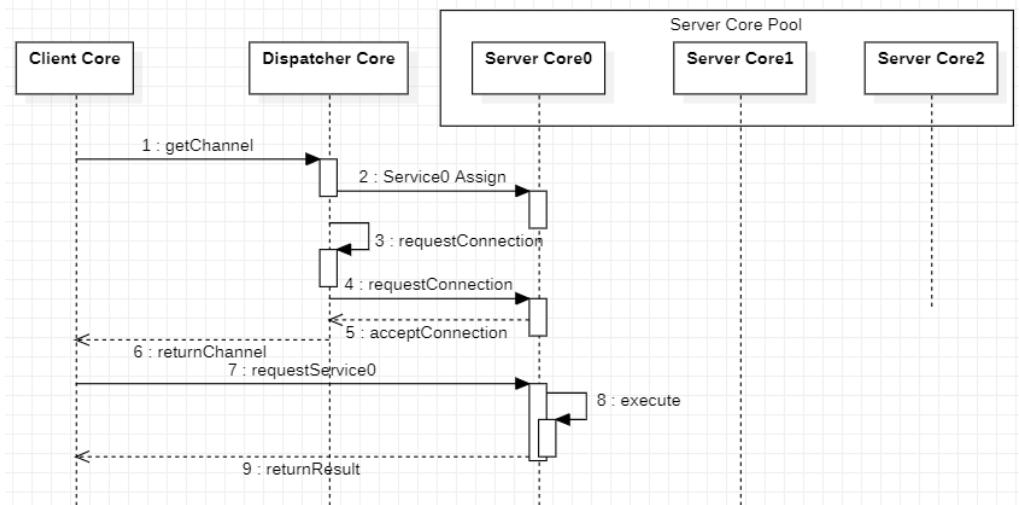


그림 42. Server Core Pool 관리.

D1.8. CA1-8 Round Robin 할당

본 후보 구조에서는 후보 구조 C1-7에서 Pool에 있는 Server Core 할당 방식에 대한 설계를 다룬다. 본 후보 구조는 가장 기본적인 Server Core 할당 방식으로 Round Robin으로 돌아가며, Client Core의 요청을 할당하는 방식이다. Dispatcher Core가 코어 할당을 위한 오버헤드는 가장 적지만, 특정 Core에게 수행 시간이 오래 걸리는 Task가 몰릴 수 있다.

D1.9. CA1-9 남은 Task가 가장 적은 Core 할당

본 후보 구조는 Dispatcher Core가 각 Server Core의 Task Queue를 확인하여, Queuing된 Task가 가장 적은 Server Core에게 Task를 할당하는 방식이다. 이로 인해, Task 수행 Latency 단축이 가능하지만, Dispatcher Core에서 각 Server Core의 Task Queue 상황을 파악해야 하는 오버헤드가 있다.

D1.10. CA1-10 Sandbox Testing 도입

후보 구조 CA1-1 ~ CA1-9의 핵심은 다수의 코어를 활용한 병렬 처리를 다룬다. 이에 대한 공통적인 Risk로 Concurrency Task로 인한 Task Fail, Task Hang, Deadlock 상황 등이 발생할 수 있다.

따라서 Concurrency Test를 가능하도록 Sandbox Testing이 가능한 Mock, Stub 모듈을 추가한다. 본 후보 구조는 성능과는 직접적인 연관은 없지만, 성능 개선을 위한 후보 구조들의 Risk를 보완하기 위한 후보 구조이다.

D1.11. CA1-11 병렬 처리 Trace 모듈 추가

본 후보 구조는 주요 병렬 처리 수행에 대한 Trace를 담당하는 모듈을 추가하는 것이다. 이를 통해, 어느 코어에서 어떤 Task 수행 시 Fail이 발생하였는지를 쉽게 파악할 수 있다. Trace에는 Core ID, 수행 시간, 그리고 주요 Task 수행 포인트에 대한 Log가 기록되어야 한다. 다만, Log 기록을 위한 성능 저하가 발생할 수 있다.

D1.12. CA1-12 Server Core간 inode 기준 탐색 병렬화

본 후보 구조는 후보 구조 CA1-6, CA1-7의 Filesystem Traverser 컴포넌트 색정 방식과 연관이 있다. 우선 해당 후보 구조 이해를 위해서는 파일 시스템 inode 구조에 대한 설명이 필요하다. 아래 그림은 파일 시스템 inode 구조를 나타낸다. 파일 시스템 상의 모든 디렉토리와 파일은 inode를 가지고 있으며, inode에는 실제 데이터가 저장되어 있는 데이터 블록 (dnode)의 위치가 저장되어 있다. 만약 해당 디렉토리/파일이 여러 개의 데이터 블록을 가지고 있다면, 모든 dnode에 대한 탐색이 필요하다. 이렇듯, 파일 시스템 Traverse를 위해서는 inode 접근과 dnode 접근이 반복적으로 이루어져야 하며, Multi-Core 병렬화를 이용할 경우, 탐색 시간을 크게 단축시킬 수 있다.

Server Core에 Task 할당 시 아래와 같은 2개의 기준을 고려할 수 있다.

- inode별 Task 할당
- dnode별 Task 할당

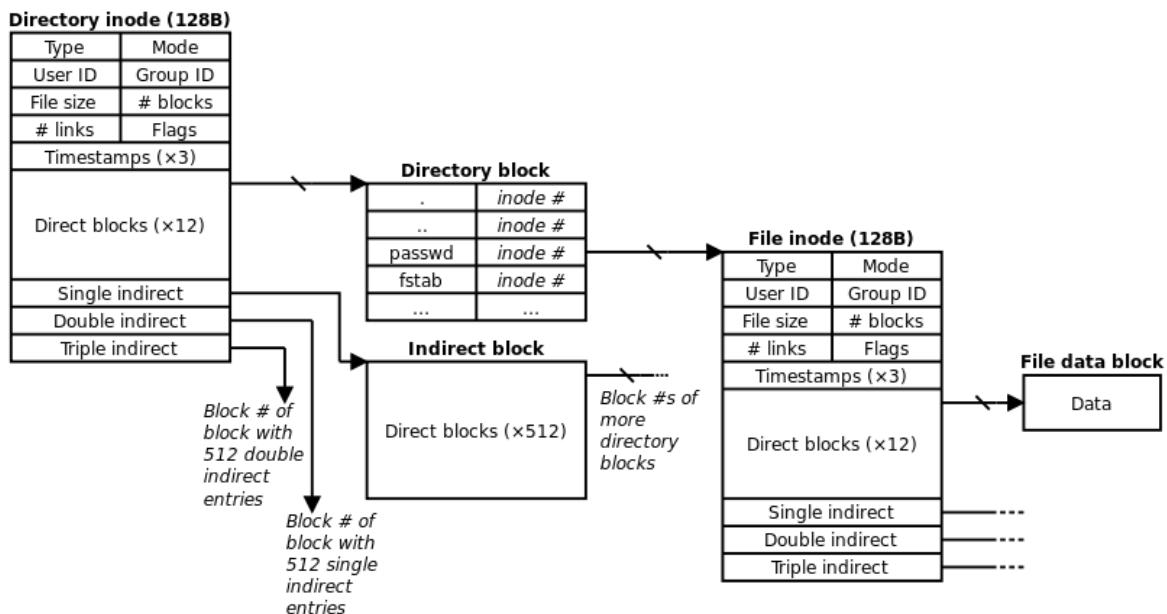


그림 43. 파일 시스템 inode 구조.

본 후보 구조와 같이 병렬 수행 시 각 Server Core에게 inode 기준으로 Task를 분배할 경우, Dispatcher Core는 디렉토리/파일마다 Server Core에서 서비스를 할당하면 된다. 이 경우, dnode 블록 기준 병렬화와 비교하여 Task 배분 Granularity가 커서 Dispatcher Core의 부하는 줄어들게 된다.

그러나 하나의 inode가 가질 수 있는 디렉토리/데이터 블록의 수가 최소 1개 ~ 최대 약 10억개 ($12 + 1024 \text{ (single indirect)} + 1024 * 1024 \text{ (double indirect)} + 1024 * 1024 * 1024 \text{ (triple indirect)}$) 개로 그 범위가 매우 넓다. 따라서 각 코어마다 수행해야 할 Task의 균등한 배분이 어려우며, 특정 Server Core로 부하가 집중될 수 있다. 이로 인해, dnode 수가 가장 많은 디렉토리/파일의 inode를 분배 받은 Server Core의 수행 시간이 하로 탐색 시간을 줄일 수 없게 된다.

D1.13. CA1-13 Server Core간 dnode 기준 탐색 병렬화

본 후보 구조 CA_1-12와 대응되는 구조로, 각 Server Core 별 dnode 탐색 Task를 할당하는 방식이다. 이 경우, 모든 Server Core는 4KB 크기의 동일한 블록 탐색을 수행하게 되어, 균등한 크기의 Task 분배가 가능해진다. 따라서 Server Core의 병렬성이 극대화되며, Filesystem Traverser 컴포넌트 수행 시간을 크게 줄일 수 있다.

그러나, Dispatcher Core에서 작은 크기의 Task를 나누어 할당하여야 하므로, Dispatcher Core의 오버헤드가 증가한다. 따라서 추후 Dispatcher Core가 병목이 될 경우, Dispatcher Core가 4KB 크기가 아닌 8KB 이상의 dnode를 할당해주는 방식으로 Dispatcher Core의 오버헤드를 조절하여야 한다.

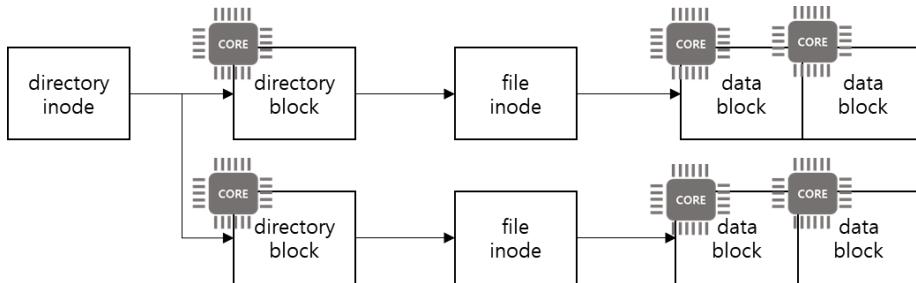


그림 44. Multi-Core 병렬화 적용 방안 (dnode 기준 탐색 병렬화).

D1.14. CA1-14 커널 이미지 파일 inode 위치 별도 저장

본 후보 구조는 커널 이미지 파일의 데이터 블록 위치를 찾기 위해 파일 시스템을 Root 디렉토리부터 Traverse하는 대신, 커널 이미지 파일의 데이터 블록을 한 번에 접근할 수 있도록, 해당 파일의 inode 위치를 별도로 저장하는 방식이다.

해당 후보 구조의 경우, 최초 한번은 파일 시스템 Traverse가 필요하다는 단점이 있지만, 그 이후 부터는 Traverse 비용 없이 커널 이미지 파일의 데이터 블록 접근이 가능하다. 또한, inode 위치의 별도 저장을 위해서는 128B 크기의 추가 저장 공간만 필요하여 오버헤드도 매우 작다.

D1.15. CA1-15 커널 이미지 압축 저장

본 후보 구조는 커널 설치 시 커널 이미지를 압축 저장하여, Boot Loader가 불러오는 커널 이미지 크기를 줄이는 방식이다. 본 후보 구조는 압축률만큼 커널 이미지 로드 시간을 절약 할 수 있지만, 압축 해제 시간이 추가로 소요된다. 압축 알고리즘 개발은 본 시스템 개발 범위 밖이나 가장 널리 쓰이는 LZO와 Snappy 알고리즘 사용을 고려해볼 수 있다. 압축률은 근소하게 LZO가 더 높고, 압축 속도는 근소하게 Snappy가 더 빠르다.

LZO와 Snappy 알고리즘의 경우, 데이터 셋에 따라 차이가 있지만 평균 70%의 압축률을 보이며,

압축 해제 속도는 약 300MB/s 정도이다. 압축 전 커널 이미지가 300MB일 경우, 압축 후 약 100MB로 줄어들게 된다. 삼성 870 EVO 기준 (읽기 성능: 약 500MB/s), 압축으로 인한 읽기 시간 단축은 0.4초이며, 압축 해제에 필요한 시간은 0.3초이다. 따라서 일반적으로 커널 이미지를 압축하여 저장할 경우, 압축을 해제하는 시간보다 커널 이미지를 읽어오는 시간의 감소가 더욱 크며, 커널 이미지 로드 시간을 줄이는데 유리하다.

본 후보 구조의 경우, 압축 해제를 위한 컴포넌트가 추가로 필요하다.

D1.16. CA1-16 커널 이미지 병렬 압축 해제

본 후보 구조는 후보 구조 CA1-15에서 다수의 Server Core를 활용하여 압축 해제를 병렬 처리한다. 이를 통해, 추가적인 압축 해제 시간 단축이 가능하며, 이로 인해, 후보 구조 CA1-15 커널 이미지 압축 저장의 효과가 더욱 커진다.

다만, 본 후보 구조의 경우, 커널 이미지를 저장 시, Storage에 커널 이미지를 분할 압축하여야 하며, 해당 정보를 Boot Loader에 전달 할 수 있도록 Config가 추가 구현되어야 한다.

D1.17. CA1-17 커널 이미지 이중화 (RAID 1)

본 후보 구조는 커널 이미지가 저장된 Storage 성능 병목이 발생할 경우 고려할 수 있다. 데이터 센터 서버의 경우 가용성 _향_상_을 위해, 부팅 드라이브 2개를 RAID 1으로 구성하는 경우가 일반적이다 (그림 2. 서버 구조. 참고). 따라서 다수의 Server Core에서 2개의 Storage에 병렬적으로 리드 명령을 내릴 경우, 커널 이미지 로드 시간을 최대 2배 단축할 수 있다 (Increase Resource Tactic).

Dispatcher Core는 Server Core에게 서비스 할당 시, (서비스 번호 % 2)와 같은 연산을 통해, Kernel Loader 컴포넌트 수행이 수행될 Storage를 지정할 수 있다. 다만, 본 후보 구조의 경우, Boot Loader가 RAID 1 구성 여부를 미리 알아야 하며, 이를 Boot Loader에 전달할 수 있도록 Config가 추가 구현되어야 한다.

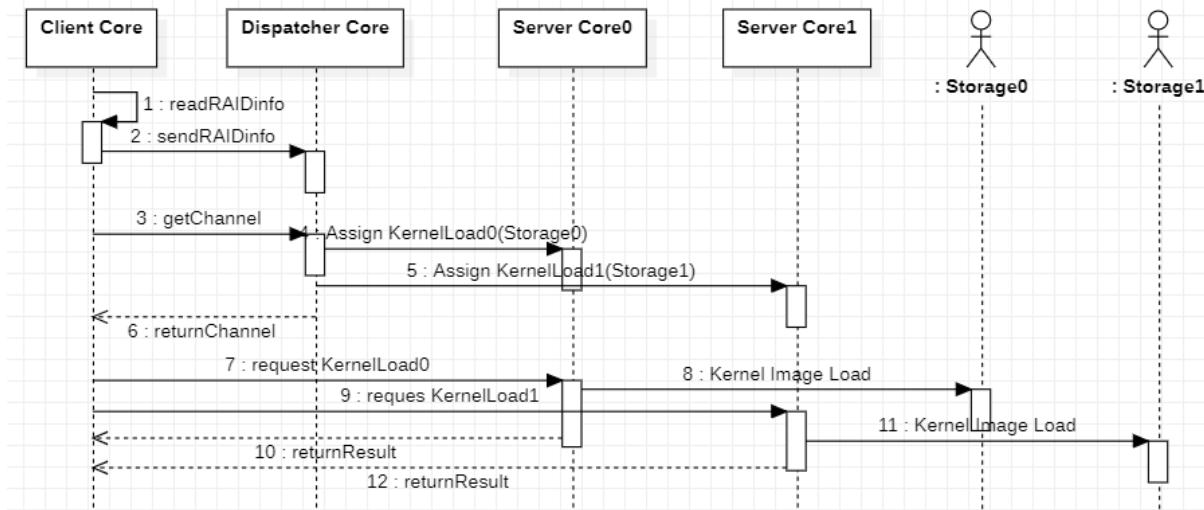


그림 45. 커널 이미지 이중화.

D1.18. CA1-18 Shell Manager 최우선 수행

Boot Loader는 부팅 과정에서 Shell Mode 진입 가능 시점부터 10초 동안 사용자가 Shell Mode 진입 키를 입력하였는지 여부를 체크하여야 한다. 따라서 Shell Mode 진입 시점을 최대한 앞당기는 것이 전체 부팅 시간 단축에 유리하다.

본 후보 구조에서는 Boot Loader가 수행되자 마자 Shell Manager를 수행한다. 데이터센터 서버의 경우, BIOS에서 POST가 완료되었으며, 따라서 Shell Manager 수행을 위한 추가적인 동작이 필요하지 않다.

D2. QA_02 디바이스 추가 및 변경 용이성

Boot Loader는 새로운 디바이스를 추가로 지원하거나, 이미 장착된 디바이스 드라이버의 변경이 용이해야 한다. 본 품질 속성은 아래와 같은 방법으로 측정된다.

- 개발 비용 (M/M) = 수정, 검증을 다시 해야 하는 모듈 / 파일의 크기(LoC)

아래 그림은 본 시스템의 도메인 모델 (그림 28) 중 품질 속성과 연관된 컴포넌트들이다. QA_02 후보 구조에서는 커널 이미지가 저장된 Storage Device와 사용자 입출력을 담당하는 IO Device의 추가 및 변경 용이성 설계 이슈에 대한 후보 구조를 상세히 기술한다.

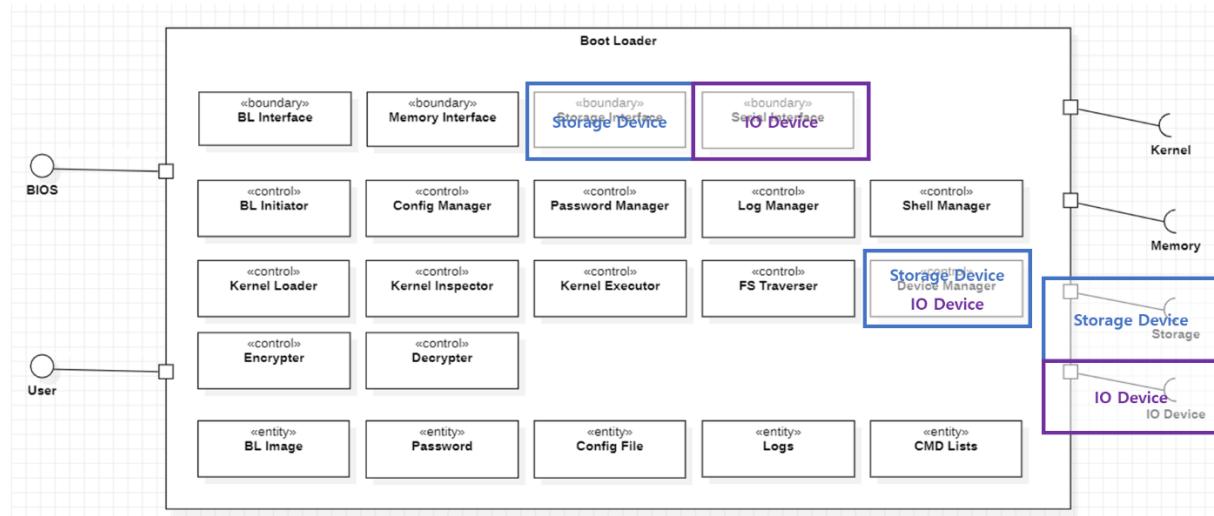


그림 46. 시스템 디바이스 종류.

아래 그림은 QA_02 디바이스 추가 및 변경 용이성 후보 구조를 요약한 그림이다.

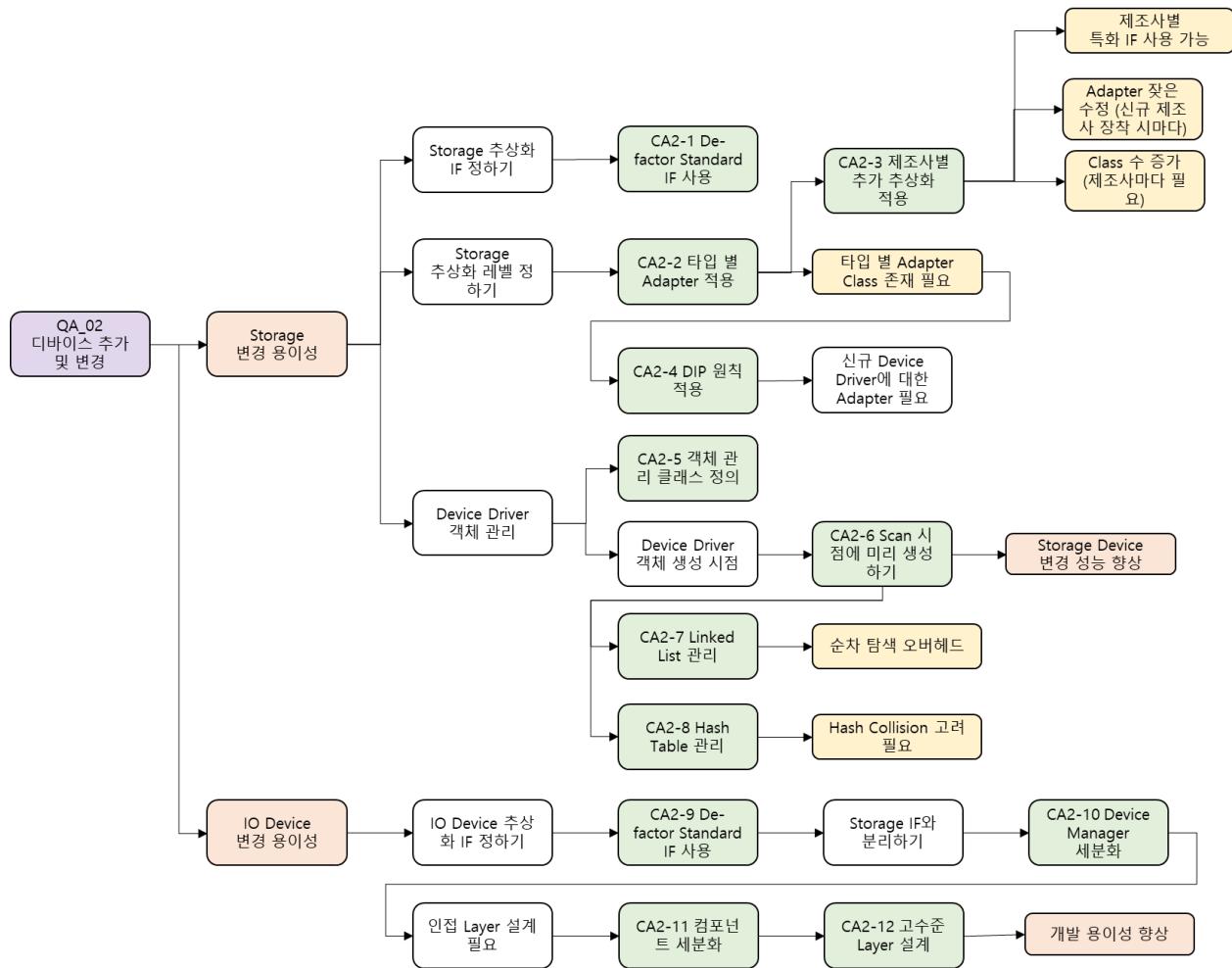


그림 47. QA_02 스토리지 장치 추가에 따른 변경 용이성 후보 구조.

D2.1. CA2-1 Storage Device 추상화를 위한 De-factor Standard 인터페이스 사용

본 후보구조에서는 도메인 모델 (그림 28) 중 Storage Interface 컴포넌트에 사용할 추상화 인터페이스를 다룬다. 우선 해당 후보 구조의 이해를 위해서는 스토리지 장치에 대한 설명이 필요하다. 아래 그림은 스토리지 장치의 Hierarchy를 나타낸다. 스토리지 장치는 다양한 타입 (ex. SATA, IDE, SCSI, NVMe)들이 존재하며, 스토리지 타입마다 각 제조사가 다양한 제품들을 출시하고 있다.

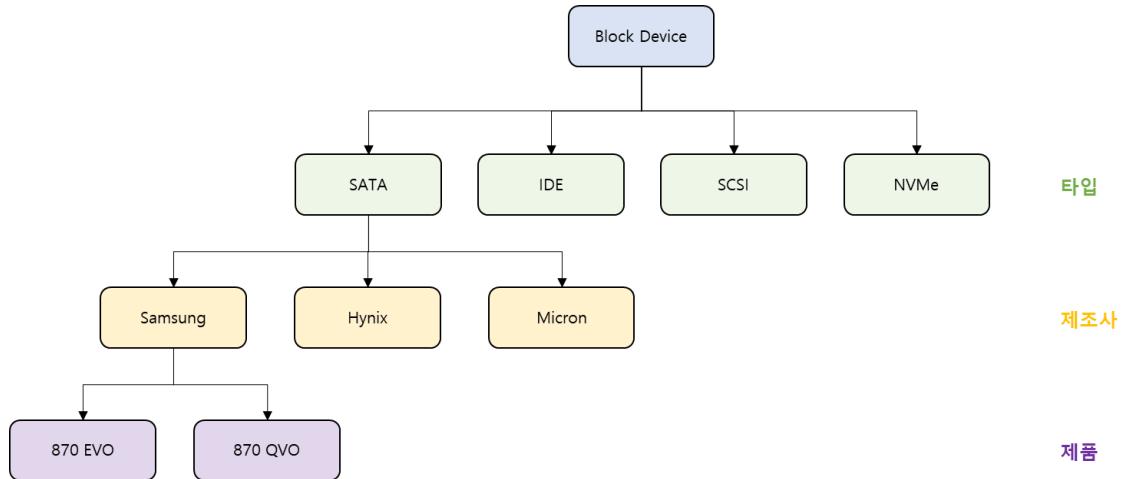


그림 48. 스토리지 장치 계층도.

아래는 Storage Device Driver들이 제공하고 있는 인터페이스를 나타낸다. 아래 표에서 알 수 있듯이, Storage Device 타입 별로 인터페이스가 조금씩 차이가 있지만, 공통적으로 init(), read(), write(), scan()을 위한 인터페이스를 제공하는 것을 알 수 있다.

	Interface
SATA	<ul style="list-style-type: none"> init_sata(int dev) sata_read(int dev, ulong blknr, ulong blkcnt, void *buffer) sata_write(int dev, ulong blknr, ulong blkcnt, const void *buffer) scan_sata(int dev)
IDE	<ul style="list-style-type: none"> ide_init() ide_read(int device, lbaint_t blknr, lbaint_t blkcnt, void *buffer) ide_write(int device, lbaint_t blknr, lbaint_t blkcnt, const void *buffer) ide_ident (block_dev_desc_t *dev_desc)
NVMe	<ul style="list-style-type: none"> int nvme_init(struct udevice *udev) nvme_blk_read(struct udevice *udev, lbaint_t blknr, lbaint_t blkcnt, void *buffer) nvme_blk_write(struct udevice *udev, lbaint_t blknr, lbaint_t blkcnt, const void *buffer) ide_ident(block_dev_desc_t *dev_desc)

본 후보 구조에서는 Storage Interface 컴포넌트를 위해 이미 De-factor Standard된 인터페이스를

채택한다. De-facto Standard 인터페이스는 기존의 Storage Device Driver를 모두 커버 가능하며, 향후 나올 Storage Device Driver로도 확장 가능할 것으로 예상된다. 따라서 추상화의 장점인 다양성 확보 및 변경 용이성 강화가 가능하다.

다만, 해당 인터페이스와 스토리지 타입 별 인터페이스에는 차이가 존재하여, 이를 위한 Adapter가 필요하다. 이는 후보 구조 CA2-2에서 다룬다.

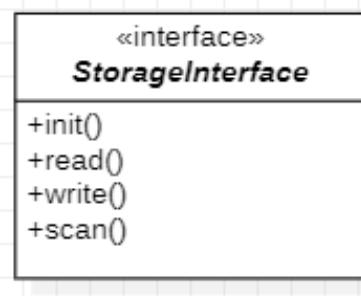


그림 49. Storage Interface 컴포넌트 추상화 인터페이스.

D2.2. CA2-2 스토리지 타입 별 Adapter 적용

본 후보 구조는 타입 별 조금씩 차이가 있는 인터페이스를 후보 구조 CA2-1에서 정의한 추상화 인터페이스로 변환하기 위해, 스토리지 타입 별 각각의 Adapter를 두는 구조이다. 이를 통해, Storage Device 변경 및 신규 Storage Device 장착 시에도 Storage Interface 컴포넌트와 이를 참조하는 Device Manager 컴포넌트가 변경되지 않는다. 다만, 본 후보 구조는 스토리지 타입 별 Adapter가 필요하기 때문에, 신규 Storage Device 장착 시, 이를 위한 신규 Adapter를 추가 구현하여야 한다.

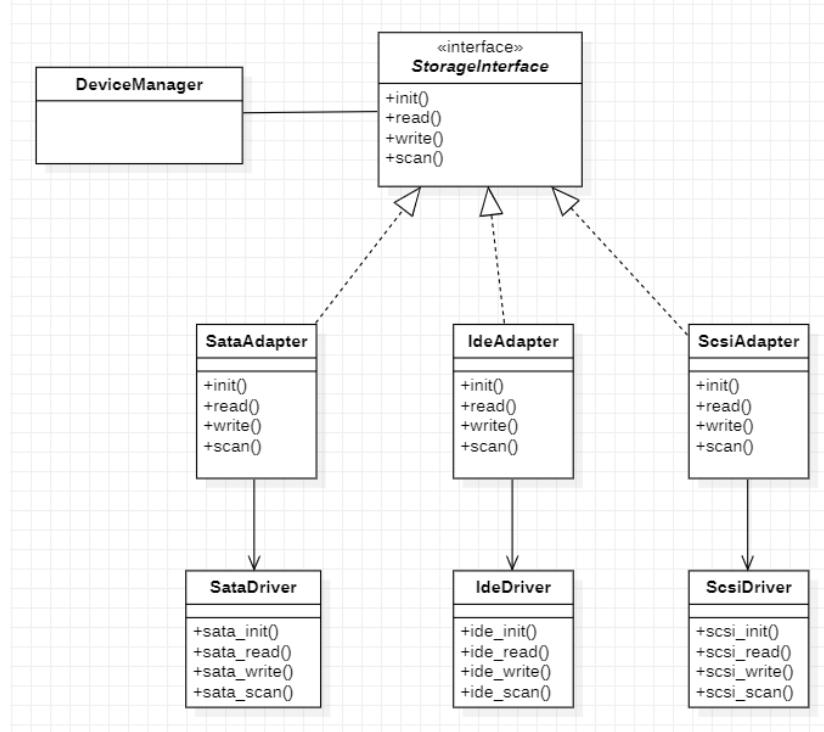


그림 50. 스토리지 타입 별 Adapter 적용.

D2.3. CA2-3 제조사별 추가 추상화

본 후보 구조는 후보 구조 CA2-2에서 제조사별 추가 추상화 인터페이스를 두는 구조이다. 예를 들어, 삼성 Storage Device의 경우, multistream()이라는 특화 인터페이스를 제공한다. 해당 인터페이스를 사용할 경우, 스토리지에 성격이 유사한 데이터를 함께 저장할 수 있어, 장기간 Read/Write 시에 성능 _향_상_ 및 수명 증가를 기대할 수 있다.

본 후보 구조는 후보 구조 CA2-2와 동일하게 Storage Device 변경 및 신규 Storage Device 장착 시에도 Storage Interface 컴포넌트와 이를 참조 Device Manager 컴포넌트가 변경되지 않는다. 다만, 본 후보 구조는 신규 제조사 스토리지 장치로의 변경 및 추가 장착할 경우, 타입 별 Adapter Class가 변경되어야 하며, 제조사에 대한 신규 인터페이스 클래스를 생성하여야 한다.

이렇듯, 본 후보 구조는 후보 구조 CA2-2와 비교하여, 변경 용이성이 낮음에도 불구하고, 본 시스템의 경우, De-factor Standard 인터페이스로 충분히 동작 가능하며, 장기간 성능 _향_상_ 및 수명 증가를 위한 제조사별 특화 인터페이스를 통해 추가적인 이점을 기대하기 어렵다. 따라서, 본 후보 구조와 같은 추가적인 추상화가 필요하지 않을 것으로 판단된다.

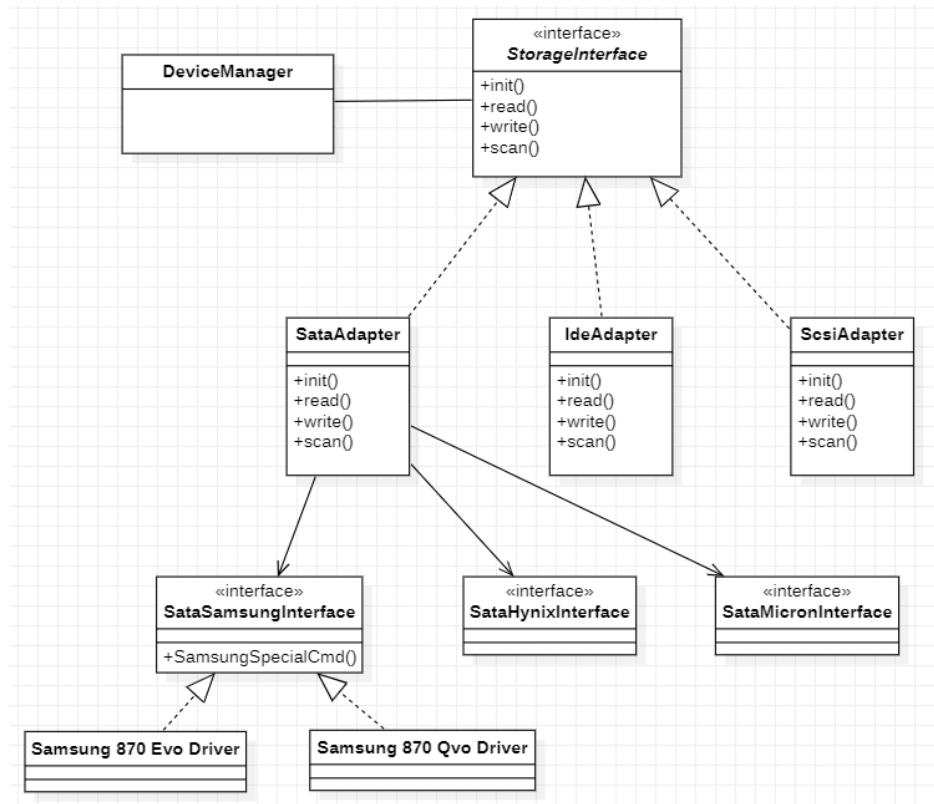


그림 51. 제조사별 추가 추상화.

D2.4. CA2-4 DIP 원칙 적용

본 후보 구조는 후보 구조 C2-2에 DIP 원칙을 적용한다. 이는 Storage Interface 컴포넌트에서 채택한 De-factor Standard 인터페이스의 경우 2000년대 이래로 변화하지 않았으며, 따라서 향후에도 변화 가능성은 거의 없다고 판단된다. 그러나 새로운 타입의 스토리지 출현이나 신규 제조사 출현 혹은 신규 제조사 특화 인터페이스의 출현은 기술의 발전에 따라 지속적으로 발생할 것으로 예상된다.

따라서 Storage Interface 컴포넌트를 고수준의 Device Manager 패키지에 포함시키고, 저수준의 Device Driver에서 이를 상속받도록 의존성 역전 원칙을 적용한다. 의존성 역전 원칙 적용을 위해서는 Device Driver의 수정이 필요하지만, 적용 후에는 디바이스 변경 및 신규 디바이스 추가 장착 시에도 고수준 컴포넌트가 변경되지 않으며, 별도의 Adapter Class의 구현도 필요 없어 변경 용이성이 가장 뛰어나다.

다행히, 현재 U-Boot나 Grub에서는 해당 후보 구조를 따르고 있다. U-Boot나 Grub은 가장 널리 사용되는 Boot Loader로 각 제조사가 자신들의 디바이스 드라이버 인터페이스를 U-Boot, Grub에 맞게 수정하였다. 이러한 배경에는 위에서 언급한 것처럼 스토리지 인터페이스의 경우 약 20년 넘게 변화가 없으며, 앞으로도 변경될 가능성이 거의 없기 때문이다. 단, 인터페이스가 다른 Device Driver의 사용을 위해서는 여전히 후보 구조 CA2-2와 같이 Adapter를 도입하여야 한다.

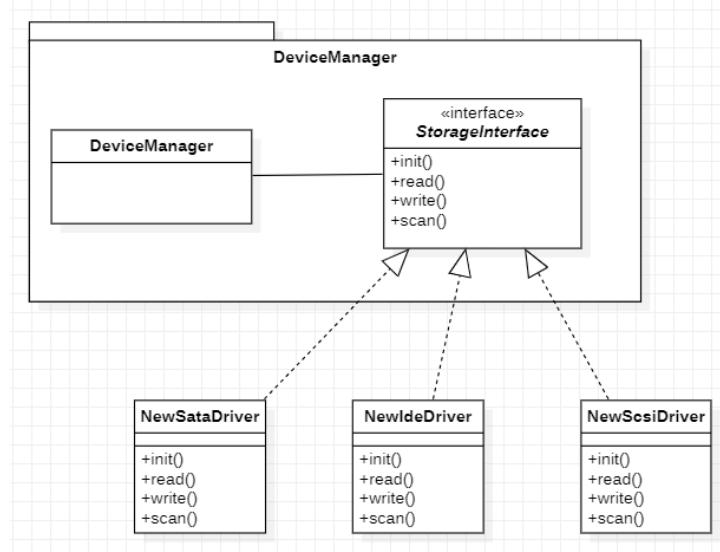


그림 52. 의존 역전 원칙 적용.

D2.5. CA2-5 Storage Device 객체 관리 클래스 정의

데이터센터에는 다수의 Storage Device들이 장착되며, 이에 대한 객체들을 관리하여야 한다. 일반적으로 하나의 서버에는 10개 ~ 40개 정도의 Storage Device가 장착된다. 이 중, Default로 사용할 Storage Device의 경우, Config File에 저장되어 있지만, 사용자가 Shell CMD를 통해 다른 Storage Device로 선택이 가능하다.

따라서 Default Storage Device 혹은 사용자가 요청한 Storage Device에 대해 객체를 생성하고, 관리하는 방안이 필요하다. 본 후보 구조에서는 아래 그림과 같이 Device Manager 컴포넌트에 객체 저장을 위한 Attribute 및 객체 추가 및 탐색을 위한 Operation들을 정의한다.

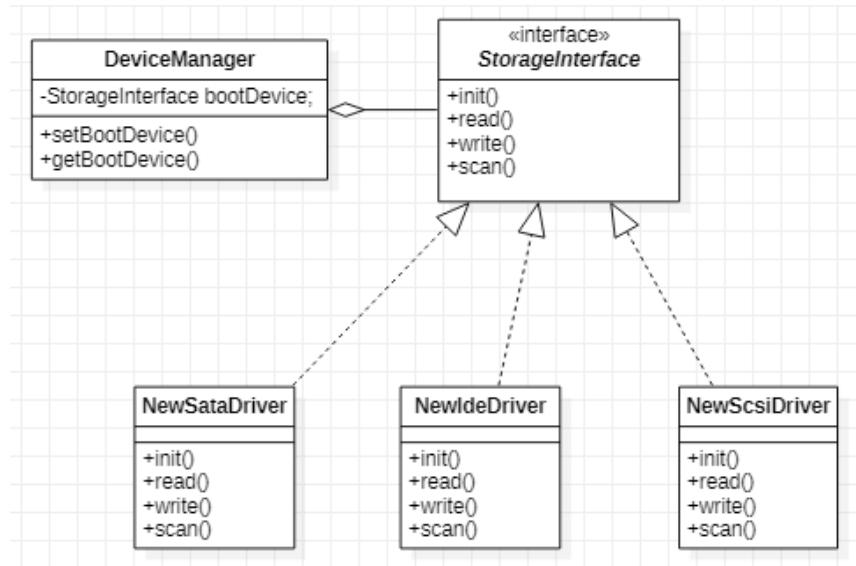


그림 53. Storage Device 객체 관리 클래스.

D2.6. CA2-6 Scan 시점에 미리 객체 생성하기

본 후보 구조는 후보 구조 CA2-5에서 성능 _향_상_을 위한 후보 구조로, Shell CMD를 통한 사용자의 Storage Device 변경 시점이 아닌 Storage Device Scan 시점에 미리 객체들을 생성한다. 이를 통해, Storage Device 변경에 대한 Shell CMD의 수행 성능이 _향_상_된다.

다만, 이 경우, 사용자가 어떠한 Storage Device로의 변경을 원할지 알 수 없기 때문에, 모든 Storage Device에 대한 객체를 미리 생성하여야 하며, 따라서 객체를 저장할 Linked List 혹은 Hash 등의 자료 구조 및 메모리 공간이 필요하다. 다행히, 데이터센터 서버에 장착되는 Service Device 수가 10개 ~ 40개임을 고려할 때, 메모리 공간 오버헤드는 거의 없을 것으로 판단된다.

다수의 객체를 저장할 자료 구조 선정에 대한 설계는 후보 구조 CA2-7과 CA2-8에서 다룬다.

D2.7. CA2-7 Linked List를 통한 Device 객체 관리

본 후보 구조는 CA2-6에서 다수의 객체를 관리하기 위한 자료 구조 설계를 다룬다. 다수의 객체들은 기본적으로 Linked List로 관리 가능하다. 단 이 경우, Scan 과정에서 생성된 신규 객체에 대한 추가에 대한 복잡도는 $O(1)$ 이나 원하는 객체 탐색 시, 순차 탐색 ($O(n)$)을 수행하여야 한다.

D2.8. CA2-8 Hash Table을 통한 Device 객체 관리

본 후보 구조는 후보 구조 CA2-7에서 Linked List를 통한 Device 객체 관리 구조와 대응되는 구조로 Linked List 대신 Hash Table을 통하여 Device 목록을 관리하는 구조이다. Hash Table 구성을 위해, Hash 키 생성 및 Collision 관리 오버헤드가 추가되지만, 탐색 복잡도는 O(1)이다.

D2.9. CA2-9 IO Device 추상화를 위한 De-factor Standard 인터페이스 사용

본 후보 구조에서는 도메인 모델 (그림 28) 중 사용자 입출력 디바이스를 위한 인터페이스인 Serial Interface 컴포넌트에 사용할 추상화 인터페이스를 다룬다. IO Device는 직렬 포트를 활용한 통신을 수행하며, 키보드, 모니터 등이 있다.

아래는 IO Device Driver가 제공하고 있는 인터페이스를 나타낸다. 아래 표에서 알 수 있듯이, IO Device와 같은 Serial Device의 경우, 이미 De-factor Standard되었으며, 더 이상 새로운 인터페이스가 추가되지 않고 있다. 따라서 본 후보 구조에서는 아래 4가지 인터페이스를 IO Device 추상화를 위해 사용한다. De-factor Standard 인터페이스는 기존의 IO Device Driver를 모두 커버 가능하며, 혹시나 신규 IO Device Driver가 나오더라도, 지원 가능할 것으로 예상된다. 따라서 추상화의 장점인 다형성 확보 및 변경 용이성 강화가 가능하다.

	Interface
IO Device	<ul style="list-style-type: none"> • open_serial(char *dev_name, int baud, int vtime, int vmin) • close_serial(int fd) • read(int fd, void *buffer, int size) • write(int fd, const void *buffer, int size)

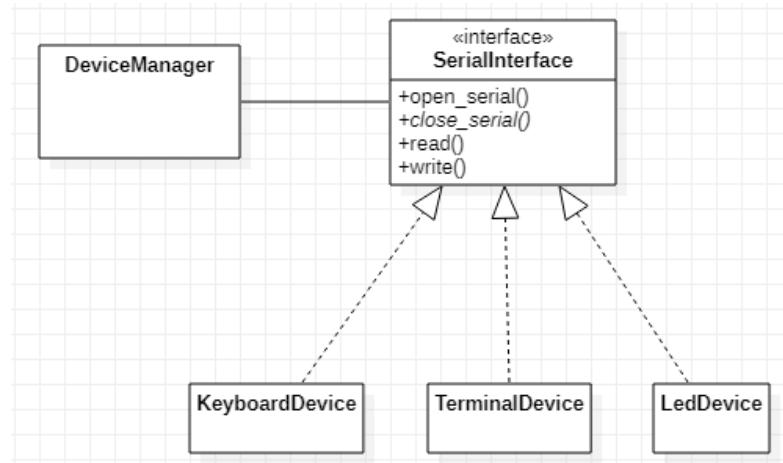


그림 54. Serial Interface 컴포넌트 추상화 API.

D2.10. CA2-10 Device Manager 세분화

후보 구조 CA2-9에서 채택한 Serial Device 추상화 인터페이스는 Storage 추상화 인터페이스와 차이가 있다. 따라서 모든 Device에 대한 단일 인터페이스로의 추상화는 불필요하다. 따라서 본 후보 구조에서는 도메인 모델 (그림 28) 중 Single Responsibility Principle을 적용하여 Device Manager를 Storage와 IO Device를 위한 컴포넌트로 세분화한다. 이를 통해, 추후, 각자의 인터페이스 변경이나 Business 로직 변경이 서로에게 영향을 미치지 않게 된다.

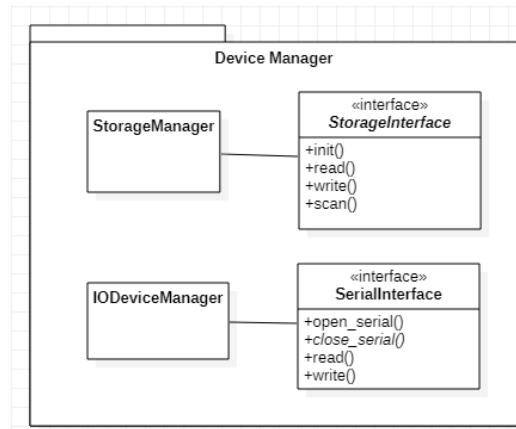
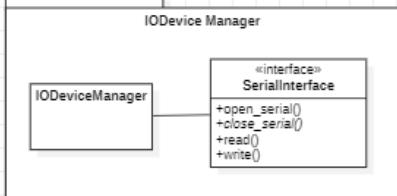


그림 55. Device Manager 세분화

D2.11. CA2-11 컴포넌트 세분화

본 후보 구조에서는 Layered style 적용에 앞서, 도메인 모델 (그림 28)에 도출된 컴포넌트들을 세분화한다. 본 후보 구조를 통해 도출된 세분화 모듈은 다음과 같다.

공통 모듈 추출	CLI Manager: Shell Manager와 Password Manager에서 사용자 입력을 받는 모듈	<pre> graph TD ShellManager[Shell Manager] --> CLIManager[CLI Manager] PasswordManager[Password Manager] --> CLIManager </pre>
변경 빈번 모듈 분리	Shell Manager: 명령어 변경 용이를 위해, 명령어 종류에 따라 별도의 처리 모듈 추가 (Boot CMD Executor, File CMD Executor, Device CMD Executor)	<pre> graph TD ShellManager[Shell Manager] --> BootCMDExecutor[Boot CMD Executor] ShellManager --> FileCMDExecutor/FileCMDExecutor[File CMD Executor] ShellManager --> DeviceCMDExecutor[Device CMD Executor] </pre>
	Config Manager: CA1-15, CA1-17 챕터 시 변경 필요 (Config File Read 모듈과 Config Setting 모듈 분리)	<pre> graph TD ConfigManager[Config Manager] --> ConfigRead[Config Read] ConfigManager --> ConfigSetting[Config Setting] </pre>
S_R_P 적용 모듈	Device Manager: Storage Manager와 IODevice Manager로 분리 (CA2-10)	<pre> graph TD StorageManager[StorageManager] --> StorageInterface["Storage Interface
+init()
+read()
+write()
+scan()"] IODeviceManager[IODeviceManager] --> SerialInterface["Serial Interface
+open_serial()
+close_serial()
+read()
+write()"] </pre>
DIP 적용 모듈	Storage Manager: Storage Interface (CA2-4)	<pre> graph TD StorageManager[Storage Manager] --> StorageInterface["Storage Interface
+init()
+read()
+write()
+scan()"] StorageManager --> StorageManager[StorageManager] </pre>

	IODevice Manager: Serial Interface (CA2-9)	
--	---	--

D2.12. CA2-12 고수준 Layer 설계

후보 구조 CA2-10에서 다른 Device Manager는 본 시스템의 경계에 해당하는 Storage 및 IO Device를 사용하기 위한 컴포넌트로, 이는 본 시스템의 하위 Layer이다. 본 후보 구조에서는 Device Layer와 'allowed to use' 관계를 갖는 컴포넌트를 판별한다. 본 후보 구조는 QA_02와 직접적인 연관은 없지만, Layered Style 채택을 통한 개발 용이성 _향_상을 위한 후보 구조이다.

Sequence Diagram에서 나타난 Device Manager를 직접 호출하는 컴포넌트는 아래와 같이 총 4개로, CA2-11을 적용하여, 고수준 Layer는 다음과 같이 설계하였다.

- Filesystem Manager, Log Manager, Shell Manager, Password Manager

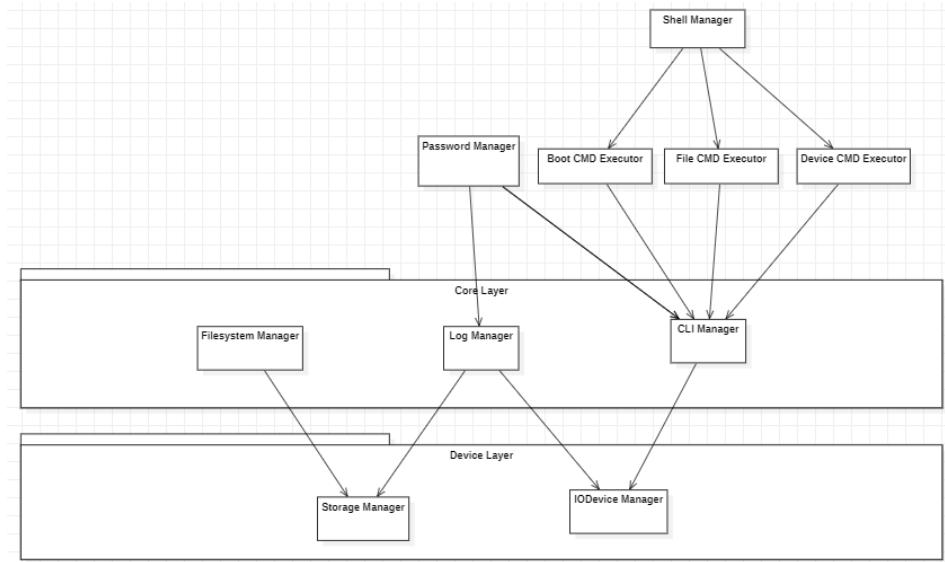


그림 56. 상위 Layer Component.

다음은 Core Layer에 대해 'allowed to use' 관계를 갖는 컴포넌트를 판별하고, 이에 대해 Layered Style 을 채택하였다. 본 시스템은 아래와 같은 4 개의 Layer 를 정의하여 개발 용이성을 높일 수 있다.

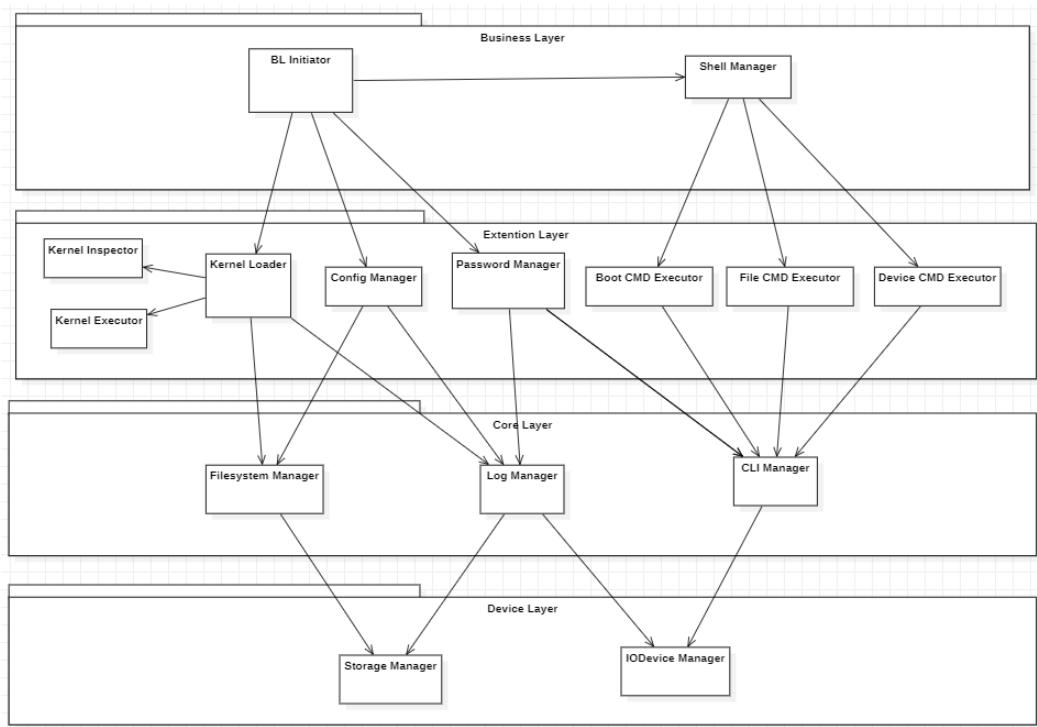


그림 57. Layered Style 적용.

D3. QA_03 Shell Command 처리 시간

본 시스템에서 Shell Command 처리 성능 향상을 위해서는 Shell Command에 대한 수행 시간을 단축하여야 한다.

- [Shell Command 수행 시간] = [Shell Command 입력 시작] – [Shell Command 결과 리턴 시작]

본 시스템의 Shell Mode는 User가 입력한 Shell Command에 대한 순차 처리만 지원하며, 여러 개의 Shell Command에 대한 병렬 처리나 Background 처리를 허용하지 않는다. 이는 Boot Loader가 제공하는 Shell Command 처리 시나리오 상 User는 선행 커맨드의 결과를 보고 다음 커맨드를 처리하는 것이 일반적이다. 또한 병렬 처리는 '단위 시간 당 처리한 Shell Command 개수'를 성능 측정 지표로 활용할 경우 필요한 성능 택틱이며, 개별 Shell Command 수행에는 오히려 성능 저하를 유발한다.

아래 그림은 Shell Manager가 User가 입력한 커맨드 처리 과정을 나타내는 Sequence Diagram이다 (그림 31). Shell Command 수행 시간을 최소화하기 위해서는 (1) CMD Parsing, (2) CMD 유효성 체크, (3) 명령어 수행의 총 3단계를 모두 최적화하여야 한다.

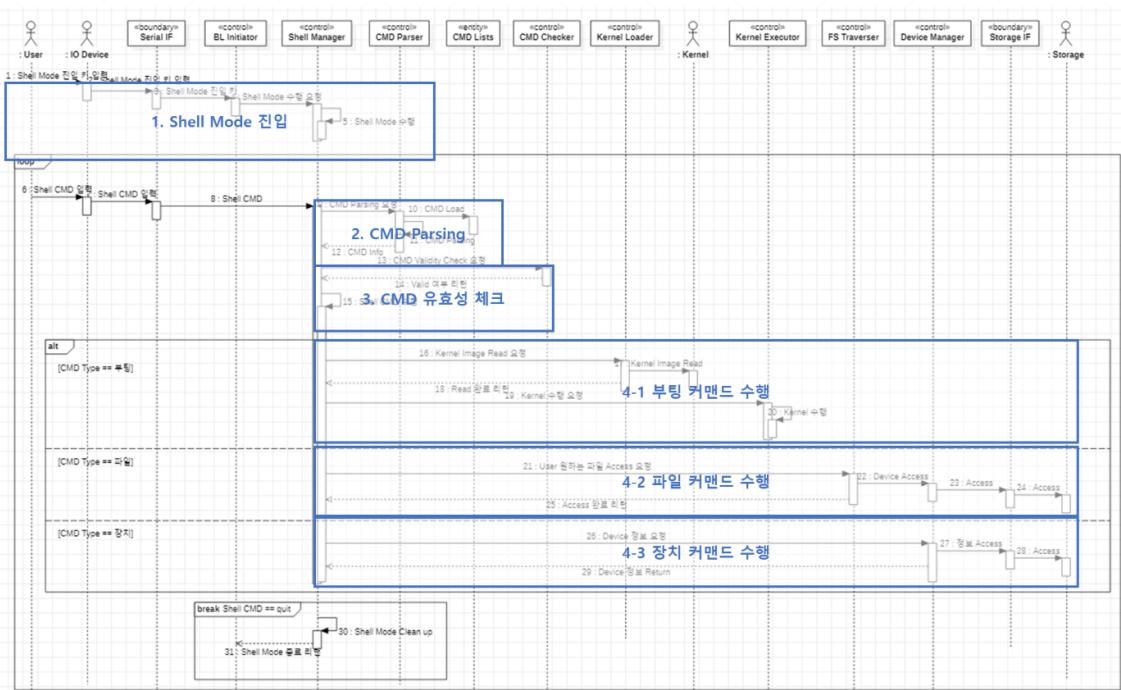


그림 58. Shell Mode 수행 Sequence Diagram.

아래 표는 Shell Mode에서 제공하는 명령어의 종류이다. Shell 명령어는 명령어별 동작 방식이 상이하여 명령어별 수행 시간 최적화가 필요하다.

분류	명령어	설명	예제
부팅 관련	root	입력한 장치를 root device로 지정	shell> root (hd0, 0) Filesystem type is ext2fs, partition type 0x83
	blocklist	입력한 파일이 저장된 blocklist 를 확인 root 명령어로 device가 지정되어 있어야 함	shell> blocklist /etc/fstab (hd0,0)1654848+1
	kernel	부팅에 사용할 커널 이미지 파일 경로를 지정 root 명령어로 device가 지정되어 있어야 함	shell> kernel /vmlinuz-2.6.18-308.el5 ro root=/dev/sda6 [Linux-bzImage, setup=0x1e00, size=0x1ceb54]
	boot	지정된 커널로 부팅 root, kernel 명령어로 device와 kernel 이 지정되어 있어야 함	shell> boot
	reboot	시스템을 재부팅 시키는 명령어	shell> reboot
	halt	시스템을 정지시키는 명령어	shell> halt
파일 시스템 관련	configfile	지정한 파일로부터 설정을 로드하는 명령어 root 명령어로 device가 지정되어 있어야 함	shell> configfile /bl/bl.conf
	Cat	지정한 파일 내용 확인 root 명령어로 device가 지정되어 있어야 함	shell> cat /etc/fstab LABEL=/ / ext3 defaults 1 1 LABEL=/boot /boot ext3 defaults 1 2
	Find	지정한 파일이 위치한 장치명을 찾아 주는 명령어	shell> find /etc/inittab (hd0,2)
장치 관련	displayapm	APM (Advanced Power Management) BIOS 정보 출력	shell> displayapm
	displaymem	물리적으로 DRAM이 설치되어 있는 시스템 주소 공간에 대한 map 표시	shell> displaymem EISA Memory BIOS Interface is present Address Map BIOS Interface is present
	geometry	지정한 device에 대한 정보를 출력	shell> geometry (hd0) drive 0x80: C/H/S = 1044/255/63, The number of sectors = 16777216, /dev/sda Partition num: 0, Filesystem type is ext2fs, partition type 0x83 Partition num: 1, Filesystem type unknown, partition type 0x82
기타	Help	Shell 명령어들에 대한 도움말을 출력	shell> help
	Clear	화면을 지우는 명령어	shell> clear

	md5crypt	Boot Loader 패스워드를 설정하는데 사용하는 MD5 암호 문자 생성기	shell> md5crypt Password: ***** Encrypted: \$1\$dSBa01\$Weux0oP2T03sW7o9DgBH71
	updatekernel	지정한 커널 이미지를 업데이트하는 명령어	shell> updatekernel /vmlinuz-2.6.18-308.el5 /vmlinuz-3.0.2-lts.el5
	Quit	Shell Mode 종료	shell> quit

아래 그림은 QA_03 Shell Command 수행 시간 관련 후보 구조를 요약한 그림이다.

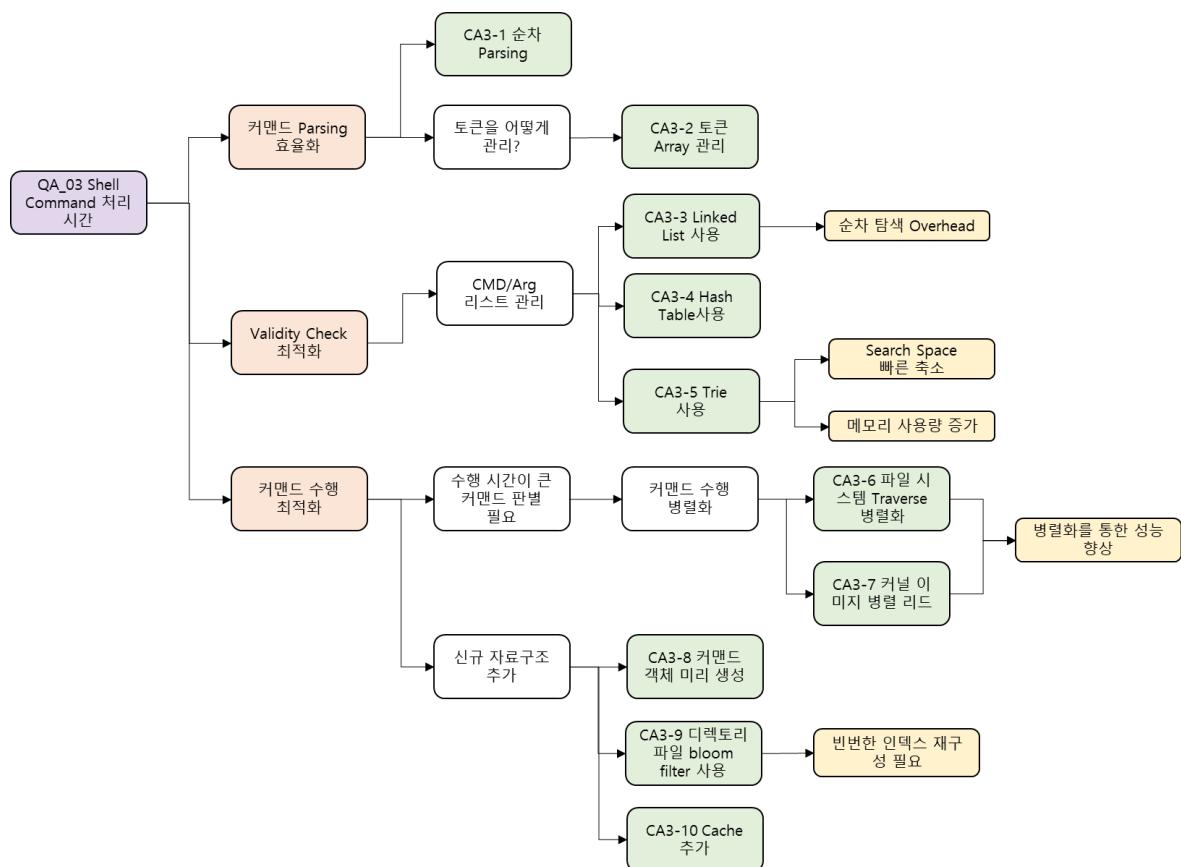


그림 59. QA_03 Shell Command 수행 시간 후보 구조.

D3.1. CA3-1 순차 Parsing

본 후보 구조는 사용자가 입력한 커맨드 문자열에 대해 구분자 기호 (공백)로 Tokenize하는 방법을 다룬다. Tokenize는 예제는 아래와 같다.

Tokenizer를 위해 IODevice Manager를 통해 입력 받은 사용자 명령어 문자열을 0번 문자부터 순차 탐색하며, 공백 여부를 확인하는 것이다. 만약 공백이 발견되면, 그 전의 모든 문자열을 하나의 토큰으로 간주한다.

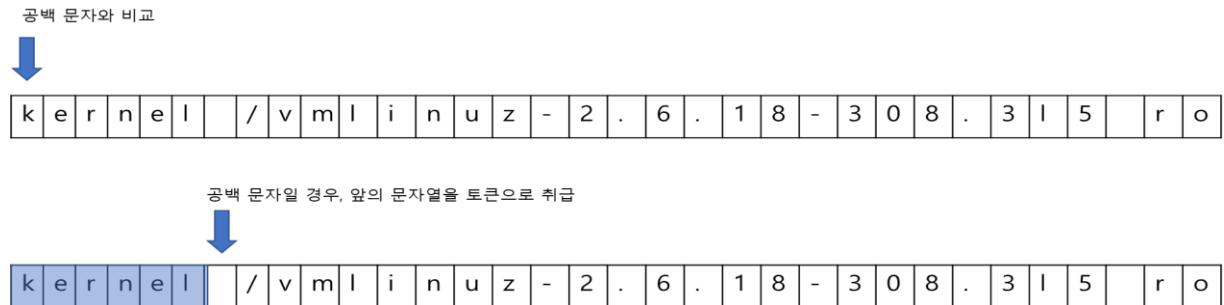


그림 60. Tokenize 예제.

Shell Command의 경우, 각 커맨드별 토큰이 최대 3개이며, 옵션이 존재하지 않다. 또한, 명령어 문자열 길이도 평균 10개 이하이다. 따라서 본 후보 구조에서는 가장 심플한 순차 탐색을 통한 Parsing을 수행한다.

참고로, Multi-Core 병렬화 등도 고려해볼 수 있지만, 오히려 병렬화를 위한 추가 오버헤드가 더 클 것으로 판단된다.

D3.2. CA3-2 Array를 통한 토큰 관리

본 후보 구조는 후보 구조 CA3-1과 연관된 후보 구조로, Tokenize 결과 생성된 토큰들을 Array로 관리한다. Array의 경우, 토큰 숫자에 대한 동적 관리가 어렵지만, 적은 수의 토큰을 빠르게 관리 할 수 있다.

참고로, Linked List나 Hash Table 등의 다른 자료 구조도 고려해볼 수 있지만, 순차 탐색이나 Hash 계산의 추가 오버헤드가 더 클 것으로 판단된다.

D3.3. CA3-3 커맨드 및 Argument 리스트 저장을 위한 Linked List 사용

사용자가 입력한 커맨드가 유효성 체크를 위해서는 Tokenize된 각 토큰에 대해 커맨드 및 커맨드 Boot Loader

Argument 리스트 탐색을 각각 수행하여야 한다. 본 후보 구조에서는 탐색이 빈번히 발생하는 커맨드 및 커맨드 Argument 리스트를 위한 자료 구조로 Linked List를 사용하는 방식이다. Linked List를 사용할 경우, 가장 Simple하지만 순차 탐색의 오버헤드가 존재한다.

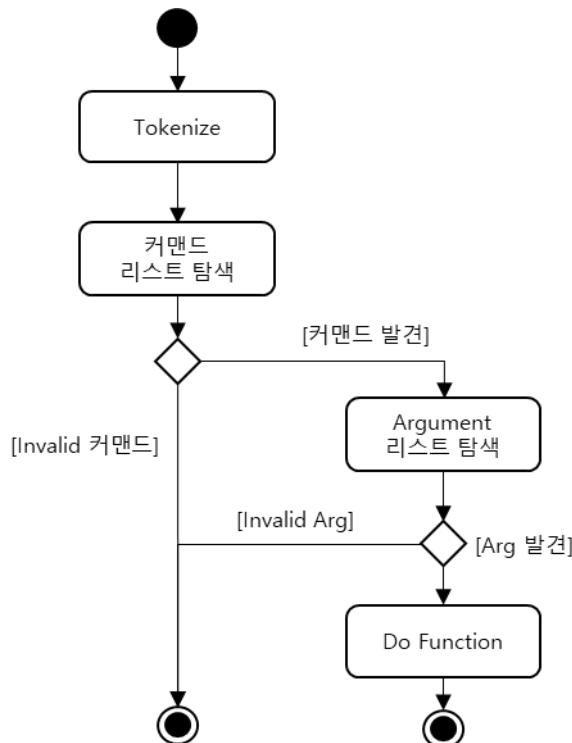


그림 61. Validity Check Activity Diagram

D3.4. CA3-4 커맨드 및 Argument 저장을 위한 Hash Table 사용

본 후보 구조는 후보 구조 CA3-4의 순차 탐색 오버헤드를 줄이기 위해, Hash Table을 사용한다. 이를 통해, 탐색 시 $O(1)$ 로 탐색이 가능하다.

D3.5. CA3-5 커맨드 및 Argument 저장을 위한 Trie 사용

Command Text의 검색을 빠르게 하기 위해서 Trie 구조를 사용하는 후보 구조이다. Trie의 탐색 속도는 $O(\text{height})$ 이다. 다만 중복된 알파벳에 대한 internal node를 생성해야 하기 때문에 문자열의

길이가 길수록 기하급수적으로 많은 메모리 용량을 차지하게 되는 단점이 있다.

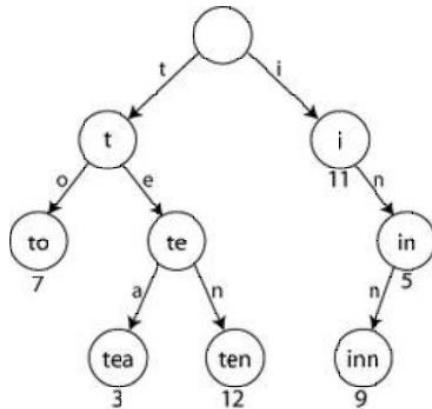


그림 62. 커맨드 Trie 구조.

D3.6. CA3-6 파일 시스템 Traverse 병렬화

Shell Command는 각 Command 종류마다 수행 방법 및 동작 방법이 매우 달라, 각 Shell Command별 수행 과정에 대한 상세 분석이 필요하다. 성능 _향_상_을 위해 상세 분석 및 후보 구조 도출이 필요한 주요 Shell Command를 아래와 같으며, Filesystem Traverse 과정과 커널 이미지 로드 과정이 포함되는 것을 알 수 있다. 참고로, 기술되지 않은 커맨드 (ex. help, clear 등)는 동작이 매우 간단하여, 추가적인 후보 구조는 설계하지 않는다.

분류	명령어	명령어 설명	상세 분석 필요 이유
부팅	blocklist	입력한 파일이 저장된 blocklist를 확인	Root 디렉토리부터 해당 파일까지 inode와 dnode를 번갈아 traverse 하는 과정 이 필요 Ex) /foo/bar Root inode -> root data block -> foo inode -> foo data block -> bar inode
부팅	boot	커널 이미지를 로드하여 부팅 수행	Root 디렉토리부터 해당 파일까지 inode와 dnode를 번갈아 traverse 하는 과정 이 필요 커널 이미지를 로드하는 과정 이 필요
파일	cat	지정한 파일 내용 확인	Root 디렉토리부터 해당 파일까지 inode와 dnode를 번갈아 traverse 하는 과정 이 필요

파일	find	지정한 파일이 위치한 장치명을 찾아주는 명령어	Root 디렉토리부터 해당 파일을 발견할 때까지, directory node Traverse 하는 과정 필요 (ex. bfs, dfs). 이로 인한, disk access 다수 발생
----	------	---------------------------	--

본 후보 구조에서는 Filesystem Traverse 성능 _향상_을 위해 Multi-Core를 사용한 병렬화 구조를 검토한다. 'blocklist', 'boot', 'cat', 'find' 커맨드 수행을 담당하는 Boot CMD Executor와 File CMD Executor는 Filesystem Traverse를 위해 Filesystem Manager 컴포넌트를 호출한다.

Filesystem Manager 컴포넌트는 Boot Loader 수행 과정에서도 동일하게 호출되며, 따라서 QA_01 후보 구조 중, Multi-Core 병렬화를 위한 후보 구조인 CA1-1 ~ CA1-14와 동일한 후보 구조를 도출할 수 있으며, 설계를 적용할 수 있다.

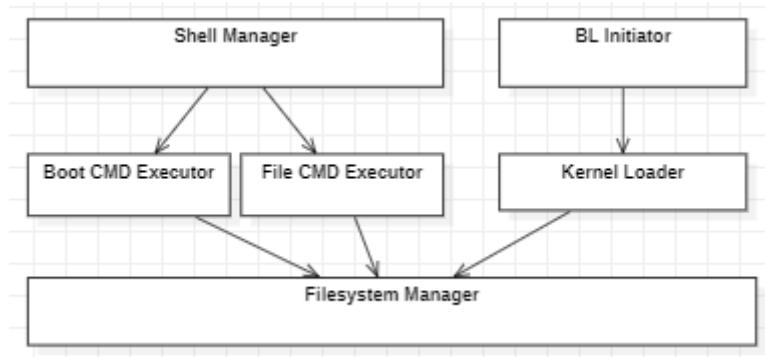


그림 63. Filesystem Traverse 수행.

D3.7. CA3-7 커널 이미지 로드 병렬화

본 후보 구조는 'boot' 커맨드 수행 시간 단축을 위해, 수행 과정에서 필요한 커널 이미지 로드를 Multi-core를 사용하여 병렬화한다. 해당 커맨드도 후보 구조 CA3-6과 동일하게 QA_01의 후보 구조인 CA1-1 ~ CA1-14를 적용할 수 있다. 추가로, 커널 이미지 로딩 최적화를 위한 추가 후보 구조 CA1-15 ~ CA1-18도 동일하게 적용 가능하다.

D3.8. CA3-8 커맨드 객체 미리 생성

본 후보 구조는 Shell Command 수행을 위한 객체를 수행 시점이 아닌, 커맨드 리스트 구축 시점

에 미리 생성한다. 이를 통해, 수행 시점에 객체 생성이 필요하지 않아, Shell Command의 수행 시간이 단축된다.

다만, 이 경우, 모든 커맨드에 대한 객체를 미리 생성하여야 하며, 객체를 저장할 추가적인 메모리 공간이 필요하다. 다행히, Shell Command는 총 17개로, 메모리 공간 오버헤드는 거의 없을 것으로 판단된다.

D3.9. CA3-9 디렉토리 파일 Bloom Filter 사용

find 명령어의 경우, 사용자가 지정한 파일을 찾을 때까지, 루트 디렉토리부터 전체 디렉토리 구조를 탐색하여야 한다. 최악의 경우, 모든 디렉토리를 다 탐색하여야 할 수 있다. 따라서 추가 디렉토리 파일에 대한 추가 인덱스를 구축할 경우, 수행 시간 향상을 크게 기대할 수 있다.

Bloom Filter는 특정 값이 해당 집합에 속하는지 여부를 검사하는데 사용되는 확률적 자료 구조로, 디렉토리 파일 내에 특정 파일의 존재 유무를 체크하기 위한 인덱스 구조로 널리 사용된다. Bloom Filter는 크기가 작아 DRAM에 로드 가능하며, 인덱스 체크 속도가 빠른 장점이 있다.

1. 원소 추가

- 추가하려는 원소에 대해서 k 개의 해시 값을 계산한 다음, 각 해시 값을 비트 배열의 인덱스로 해서 대응하는 비트를 1로 세팅한다.

2. 원소 검사

- 검사하려는 원소에 대해서 k 개의 해시 값을 계산한 다음, 각 해시 값을 비트 배열의 인덱스로 해서 대응하는 비트를 리드한다. k 개의 비트가 모두 1인 경우 원소는 집합에 속한다고 판단하고, 그렇지 않은 경우 속하지 않는다고 판단한다.

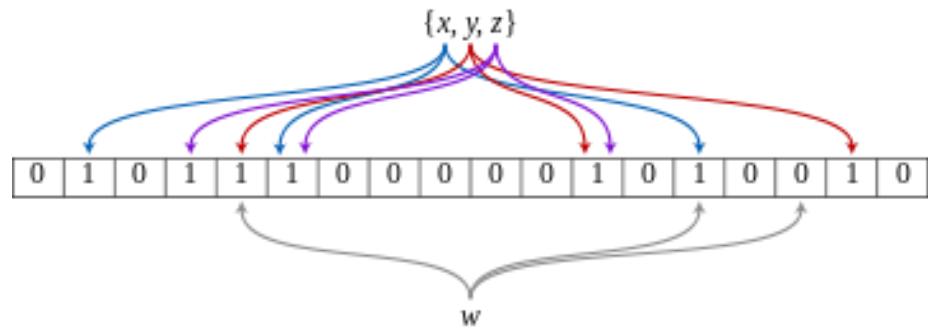


그림 64. Bloom Filter 예제.

그러나 딕렉토리가 변경될 경우, 이에 대한 Bloom Filter를 재구축이 필요하다는 단점이 있다.

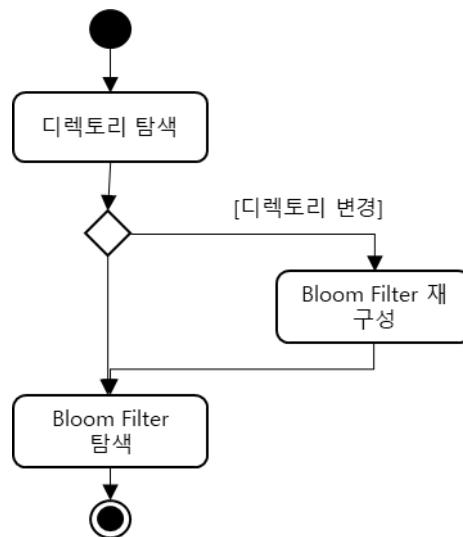


그림 65. Bloom Filter 재구성 Activity Diagram.

D3.10. CA-10 Cache 도입

Caching은 자주 사용되는 Command를 위한 전략이다. 예를 들어, 'find' 명령어가 여러 번 사용될 경우, 이를 Caching하여, 커맨드 탐색 시간 및 Filesystem Traverse 시간을 단축할 수 있다. 그러나 Shell Mode에서 커맨드가 반복 사용되지 않는다면, 오히려 Cache 공간 낭비 및 Cache 검색 오버헤드가 증가하게 되는 단점이 있다.

D4. QA_04 커널 이미지 로딩 가용성

커널 이미지 로딩 가용성 품질 속성의 측정 방법은 아래와 같다.

- [정상 복구 시간] = [Recover 완료 시각] – [Fail 발생 시각]

정상 복구 시간을 단축하기 위해서는 (1) Fail을 빨리 감지하여 복구에 필요한 비용을 줄여야 하며, (2) 복구를 빨리하여야 하며, (3) 사용자 개입이 필요한 복구의 경우, 개입 시간 단축을 위해 사용자에게 빨리 알려야 한다. 위의 3가지 측면에서 설계 이슈를 해결하기 위해 후보 구조를 검토한다.

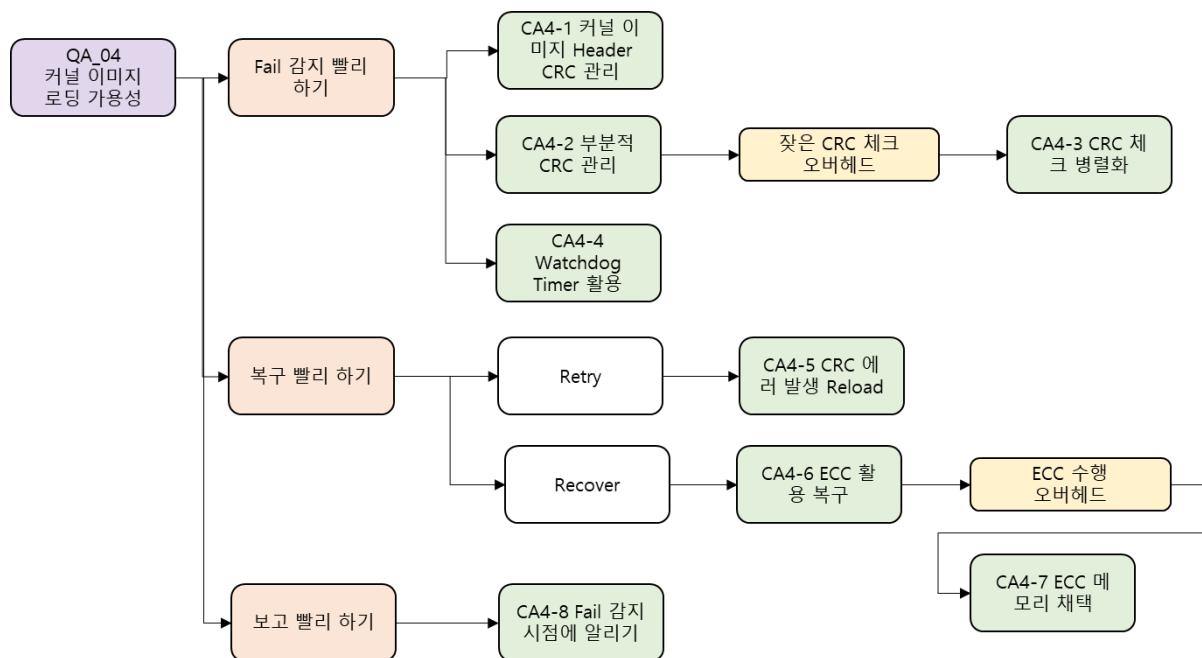


그림 66. QA_04 커널 이미지 로딩 가용성 후보 구조.

D4.1. CA4-1 커널 이미지 파일 Header CRC 관리

CRC (Cyclic Redundancy Check)는 데이터에 대한 오류 여부 검출을 위해 널리 쓰이는 방법이다. 본 후보 구조는 커널 이미지 파일 Header CRC를 통해, 파일 Header에 발생한 Fail을 빨리 감지한다.

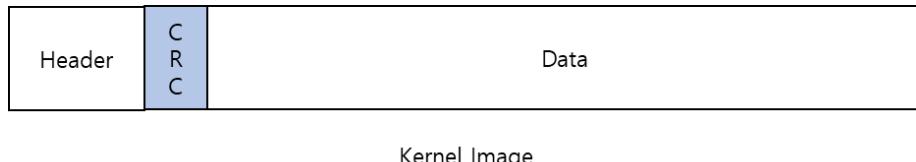


그림 67. 커널 이미지 Header CRC 관리.

D4.2. CA4-2 커널 이미지 부분적 CRC 관리

본 후보 구조는 전체 커널 이미지에 대한 하나의 CRC가 아닌 부분 이미지에 대한 각각의 CRC를 관리하는 방법이다. 이 경우, 전체 커널 이미지를 로딩하지 않고도, 커널 이미지의 특정 부분에 발생한 Fail을 빨리 감지할 수 있다. 다만, 커널 이미지에 대한 잣은 CRC 체크가 필요하다.

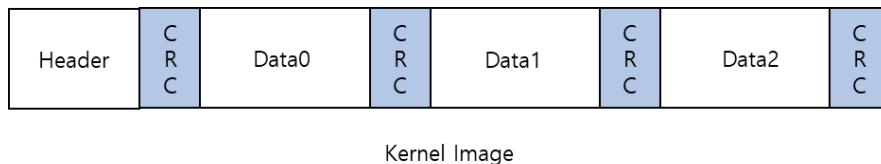


그림 68. 커널 이미지 부분적 CRC 관리.

D4.3. CA4-3 CRC 체크 병렬화

본 후보 구조는 후보 구조 CA4-2의 잣은 CRC 체크 오버헤드를 줄이기 위한 방법이다. Multi-Core를 활용하여, 부분 이미지에 대한 CRC 체크를 병렬화 함으로써, 수행 시간을 단축할 수 있다. QA_01에서 커널 이미지 로드 병렬화를 위한 후보 구조들과 동일하게, 다수의 Server Core를 활용하여 커널 이미지 로드 후, 각 Server Core에서 로드한 이미지에 CRC 체크를 수행할 수 있다.

D4.4. CA4-4 Watchdog Timer 활용

본 후보 구조는 부팅 과정에서 예상하지 못한 장애로 인한 Hang을 대응하기 위한 Watchdog Timer를 활용한다. BL Initiator에서 부팅 시작 과정에서 Watchdog Timer를 설정하고, 특정 시간 이후에도 부팅이 완료되지 않는다면, 장애 상황에 의한 Hang으로 간주하기 위함이다. 다만, Hang이 아닌 복구 진행 중일 수도 있으므로, Watchdog Timer 시간을 넉넉하게 설정하거나, 각 주요

동작마다 별도의 Timer를 설정하는 방식을 사용할 수도 있다.

D4.5. CA4-5 CRC 에러 발생 시 Reload

본 후보 구조는 CRC 에러 발생 시, 에러가 발생한 부분을 Storage Device로부터 다시 로드 한다. 이로 인해, Storage에서 DRAM으로 Kernel Image를 읽어 오는 과정에서 발생한 장애에 대해 사용자의 대응없이 복구할 수 있으며, 복구 시간을 줄일 수 있다.

단, Reload 과정으로도 복구 불가능할 경우, 오히려 복구 시간을 크게 증가시키며, 사용자 개입 시점을 늦추게 된다.

D4.6. CA4-6 ECC 활용 복구

본 후보 구조는 해밍 코드와 같은 ECC를 활용한 에러를 복구한다. ECC 수행 시간이 필요하지만, Storage Device에서 다시 로드 하는 것보다 더 빠르게 장애를 복구할 수 있다.

D4.7. CA4-7 ECC 메모리 채택

본 후보 구조는 후보 구조 CA4-6의 ECC 계산 시간 오버헤드를 줄이기 위해, ECC 기능을 제공하는 메모리를 채택한다. 이 경우, DRAM에서 자동으로 ECC를 수행하기 때문에, 별도의 ECC 계산 시간이 필요 없게 된다. 참고로, 일반적으로 서버급 메모리 (ex. ECC Registered DRAM)은 ECC 기능을 제공한다.

D4.8. CA4-8 Fail 감지 시점에 알리기

본 후보 구조는 사용자 개입을 빨리 하기 위해, Fail 감지 시점에 미리 알린다. 이는 자동으로 시스템이 수행하는 Sequence가 실패하였을 경우, 미리 사용자에게 알림으로써, 사용자 장애 인지 시점을 앞당길 수 있게 된다.

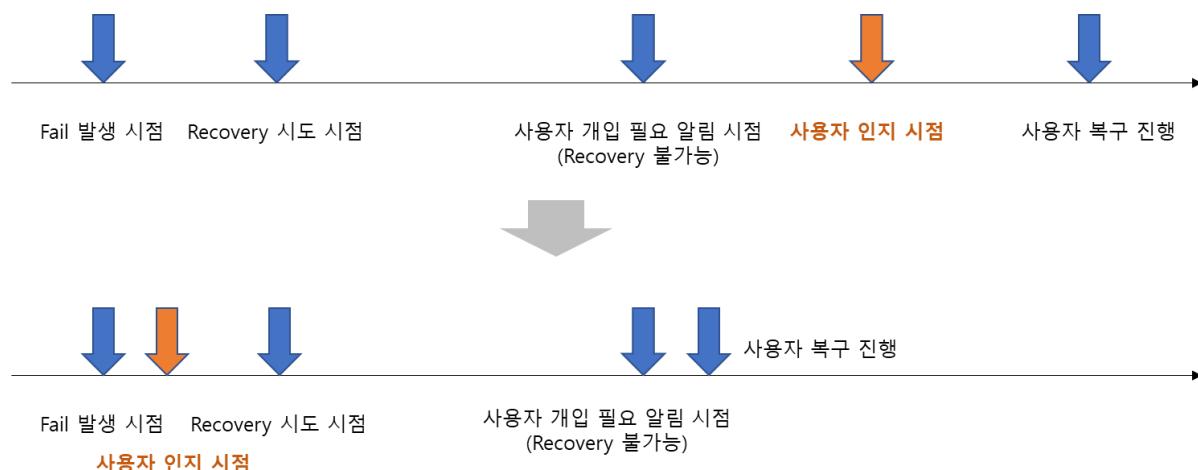


그림 69. Fail 감지 시점에서 미리 알리기.

D5. QA_05 Shell Command 추가 및 확장 용이성

Boot Loader는 새로운 Shell Command에 대한 추가 및 확장이 용이해야 한다. 본 품질 속성은 아래와 같은 방법으로 측정된다.

- 개발 비용 (M/M) = 수정, 검증을 다시 해야 하는 모듈 / 파일의 크기(LoC)

아래 그림은 본 시스템의 도메인 모델 (그림 28) 중 품질 속성과 연관된 컴포넌트들이다. QA_05 후보 구조에서는 Shell Manager, CMD Parser, CMD Checker 컴포넌트에 대한 확장 및 변경 용이성 설계 이슈에 대한 후보 구조를 상세히 기술한다.

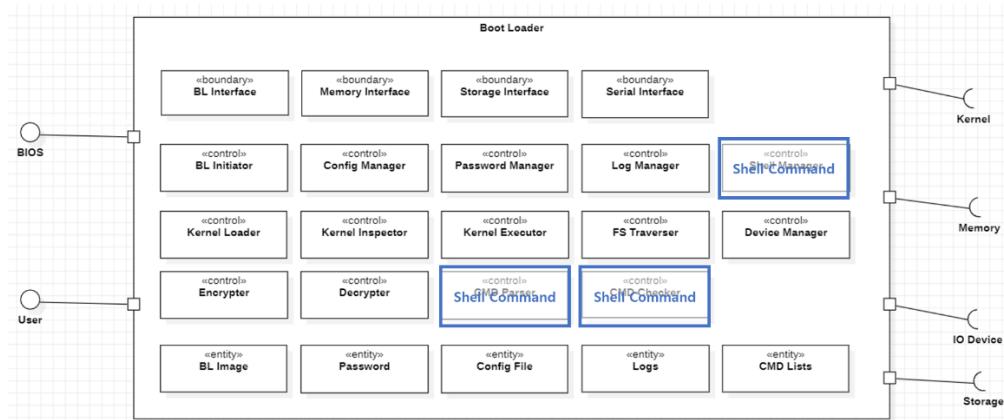


그림 70. Shell Command 관련 컴포넌트.

아래 그림은 QA_05 Shell Command 추가 확장 용이성 후보 구조를 요약한 그림이다.

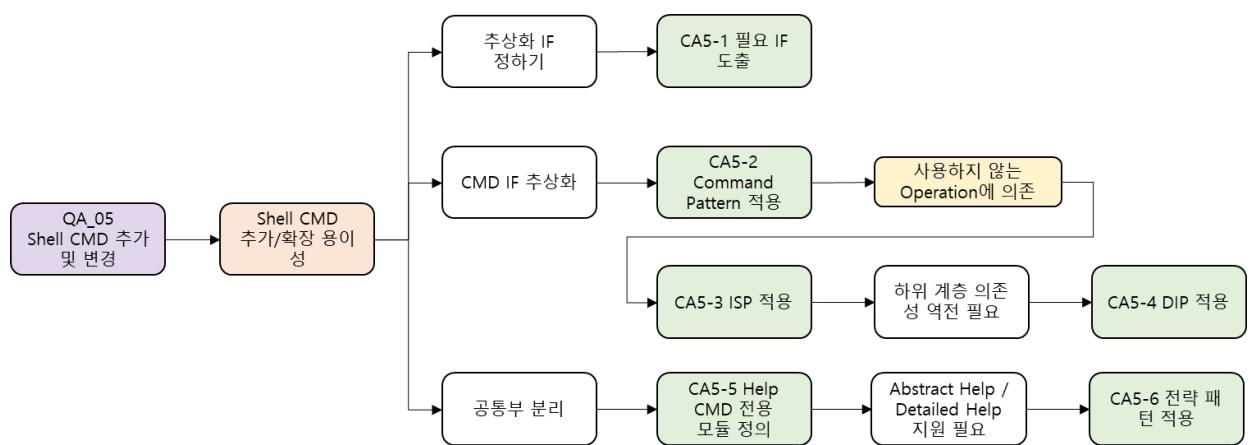


그림 71. Shell Command 추가/확장 용이성 후보 구조.

D5.1. CA5-1 Shell Command 필요 IF 도출

본 후보 구조에서는 Shell Command 인터페이스 추상화에 필요한 인터페이스를 설계한다. 이를 위해 우선 각 커マン드 별 제공해야 할 기능들을 아래와 같이 도출하였다.

- Valid Check: Tokenize 결과가 Valid한지 여부 Check
- Execute: 커マン드 실행
- Help: 각 커マン드에 대한 도움말 제공

본 후보 구조에서는 도출된 기능을 바탕으로 아래와 같이 인터페이스를 정의한다. 이를 통해, 추상화의 장점인 다형성 확보 및 변경 용이성 강화가 가능하다.

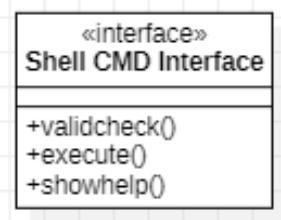


그림 72. Shell Command Interface 추상화 인터페이스.

D5.2. CA5-2 Command Pattern 적용

본 후보 구조에서는 CA5-1에서 정의한 Shell CMD Interface 컴포넌트를 사용한 Command Pattern을 적용한다. Command Pattern은 특정 객체에 대한 커맨드를 객체화하여 커맨드 객체를 필요에 따라 처리하는 패턴이다. 이를 통해, 신규 Shell Command 추가 및 기존 Shell Command 확장 시에도 CMD Executor 컴포넌트, CMD Checker 컴포넌트, Helper 컴포넌트가 변경되지 않는다.

참고로, Helper 컴포넌트 분리에 대해서는 후보 구조 CA5-5에서 다룬다.

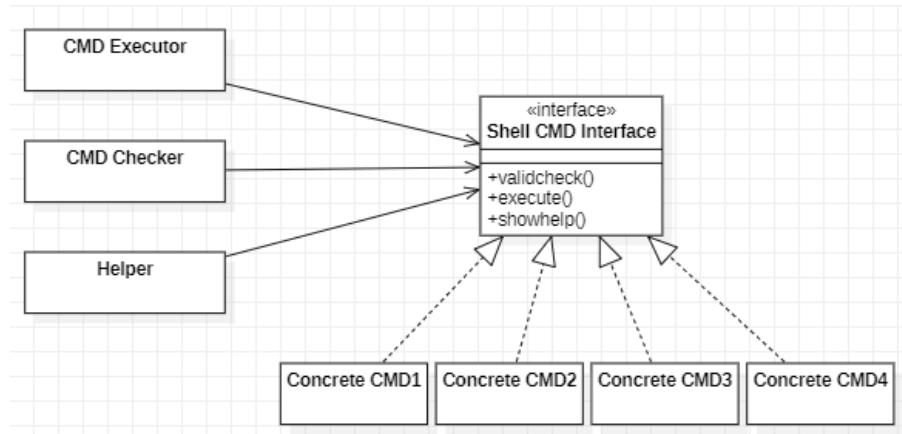


그림 73. Command Pattern 적용.

D5.3. CA5-3 ISP (Interface Segregation Principle) 적용

본 후보 구조에서는 후보 구조 CA5-2에서 Command Pattern 적용 결과, 클라이언트 컴포넌트들이 자신이 의존하지 않음에도 불구하고, 인터페이스를 참조하게 된다. 예를 들어, CMD Executor는 execute Operation만 의존하지만, validcheck(), showhelp() 인터페이스도 함께 참조하게 된다. 따라서 본 후보 구조에서는 ISP를 적용하여, 큰 덩어리의 인터페이스를 구체적이고 작은 단위의 인터페이스로 분리함으로써, 클라이언트 컴포넌트들이 꼭 필요한 Operation만 이용할 수 있게 된다.

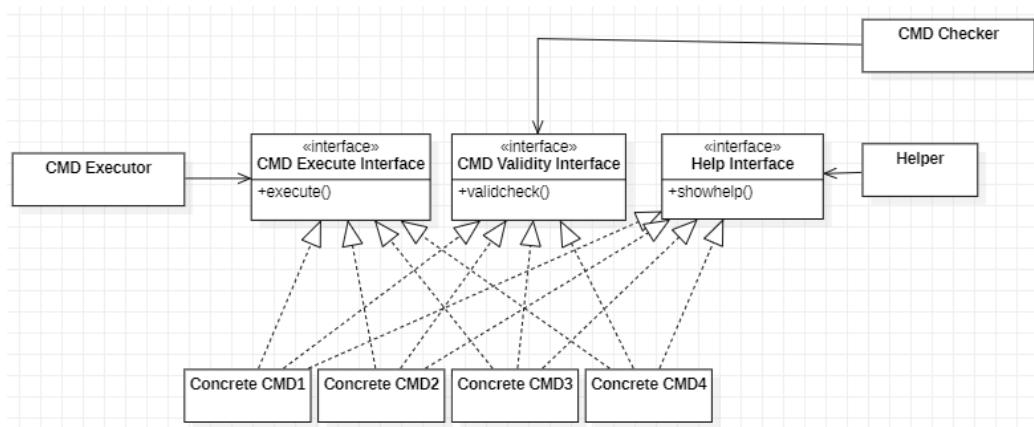


그림 74. ISP 적용.

D5.4. CA5-4 DIP (Dependency Inversion Principle) 적용

본 후보 구조에서는 DIP를 적용하여, 하위 계층에 대한 의존성을 역전시킨다. 이를 통해, Shell Command를 위한 각각의 인터페이스는 고수준의 컴포넌트 패키지에 포함됨으로써, 신규 커맨드 추가 및 확장 시, 고수준 컴포넌트가 변경되지 않는다.

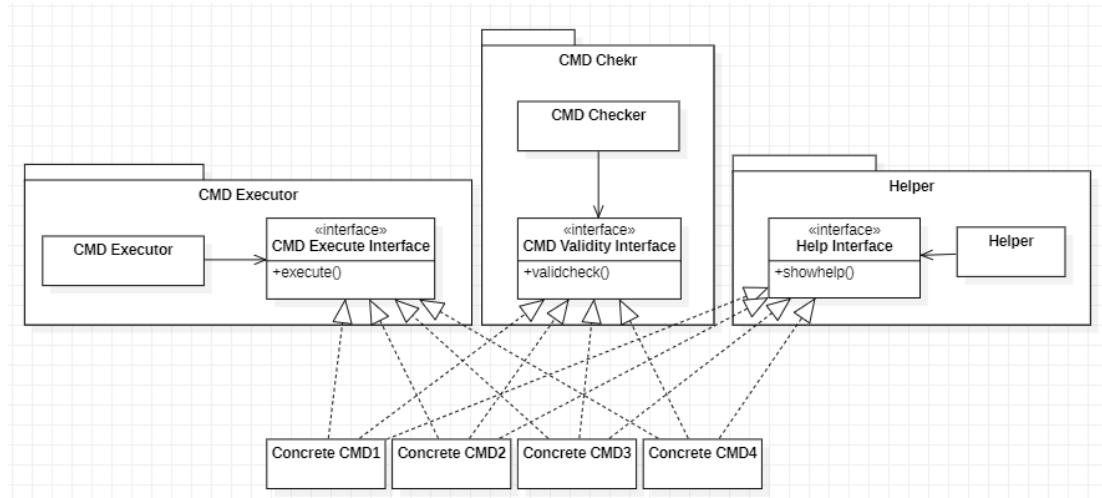


그림 75. DIP 적용.

D5.5. CA5-5 Help CMD 전용 모듈 정의

본 후보 구조에서는 각 커맨드 수행에 있어, 공통인 Help 제공 모듈을 분리한다. 이를 통해, 각 모듈은 S_R_P 원칙을 따르게 되어 응집성을 높이고, 서로의 모듈이 변경되더라도 영향을 미치지 않게 된다.

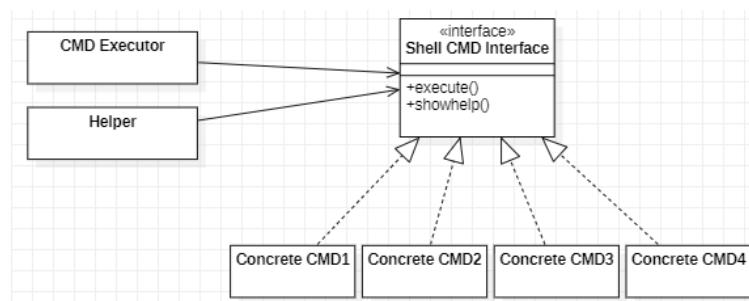


그림 76. Help 모듈 분리.

D5.6. CA5-6 전략 패턴 정의

도움말 출력에 있어, 일반적으로 간단한 도움말과 좀 더 상세한 도움말을 제공한다. 따라서 본 후보 구조에서는 사용자의 요청에 따라 도움말의 상세 여부를 선택할 수 있게, 전략 패턴을 적용한다. 이를 통해, Helper 모듈의 변경없이, 사용자가 요청한 도움말을 제공 가능하다.

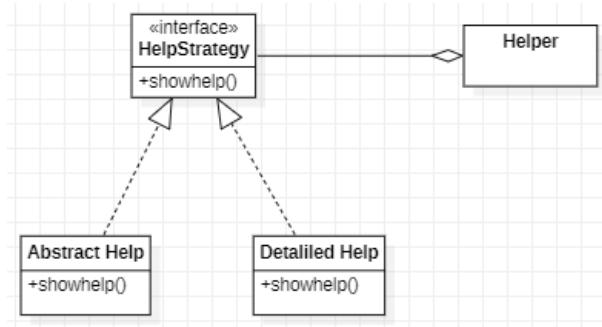


그림 77. Help 전략 패턴 적용.

E. 후보 구조 평가

No.	Description	채택 여부
CA1-1	Client/Server Core 도입	채택
CA1-2	Dispatcher Core 추가 도입	채택
CA1-3	코어 간 Shared Memory를 통한 통신	미채택
CA1-4	코어 간 Task Queue를 통한 통신	채택
CA1-5	Client Core와 비동기 수행 컴포넌트를 Server Core에 할당	채택
CA1-6	단일 컴포넌트를 다수의 Server Core에 할당	채택
CA1-7	Server Core Pool 관리	채택
CA1-8	Round Robin 할당	미채택
CA1-9	남은 Task가 가장 적은 Core 할당	채택
CA1-10	Sandbox Testing 도입	채택
CA1-11	병렬 처리 Trace 모듈 추가	미채택
CA1-12	Server Core간 inode 기준 탐색 병렬화	미채택
CA1-13	Server Core간 dnode 기준 탐색 병렬화	채택
CA1-14	커널 이미지 파일 inode 위치 별도 저장	채택
CA1-15	커널 이미지 압축 저장	채택
CA1-16	커널 이미지 병렬 압축 해제	채택
CA1-17	커널 이미지 이중화 (RAID 1)	채택
CA1-18	Shell Manager 최우선 수행	채택
CA2-1	Storage Device 추상화를 위한 De-factor Standard IF 사용	채택
CA2-2	스토리지 타입 별 Adapter 적용	채택
CA2-3	제조사별 추가 추상화	미채택
CA2-4	DIP 원칙 적용	채택
CA2-5	Storage Device 객체 관리 클래스 정의	채택
CA2-6	Scan 시점에 미리 객체 생성하기	채택
CA2-7	Linked List를 통한 Device 객체 관리	미채택
CA2-8	Hash Table을 통한 Device 객체 관리	채택
CA2-9	IO Device 추상화를 위한 De-factor Standard IF 사용	채택
CA2-10	Device Manager 세분화	채택
CA2-11	컴포넌트 세분화	채택
CA2-12	고수준 Layer 설계	채택
CA3-1	순차 Parsing	채택
CA3-2	Array를 통한 토큰 관리	채택

CA3-3	커맨드 및 Argument 리스트 저장을 위한 Linked List 사용	미채택
CA3-4	커맨드 및 Argument 리스트 저장을 위한 Hash Table 사용	채택
CA3-5	커맨드 및 Argument 리스트 저장을 위한 Trie 사용	미채택
CA3-6	파일 시스템 Traverse 병렬화	채택
CA3-7	커널 이미지 로드 병렬화	채택
CA3-8	커맨드 객체 미리 생성	채택
CA3-9	디렉토리 파일 Bloom Filter 사용	미채택
CA3-10	Cache 도입	미채택
CA4-1	커널 이미지 파일 Header CRC 관리	채택
CA4-2	커널 이미지 부분적 CRC 관리	채택
CA4-3	CRC 체크 병렬화	채택
CA4-4	Watchdog Timer 활용	채택
CA4-5	CRC 에러 발생 시 Reload	미채택
CA4-6	ECC 활용 복구	미채택
CA4-7	ECC 메모리 채택	채택
CA4-8	Fail 감지 시점에 알리기	채택
CA5-1	Shell Command 필요 IF 도출	채택
CA5-2	Command Pattern 적용	채택
CA5-3	ISP 적용	채택
CA5-4	DIP 적용	채택
CA5-5	Help CMD 전용 모듈 정의	채택
CA5-6	전략 패턴 정의	채택

E1. QA_01 Boot Loader 수행 시간

Boot Loader 수행 시간 관련 후보 구조 선정 결과를 요약하면 아래 그림과 같다.

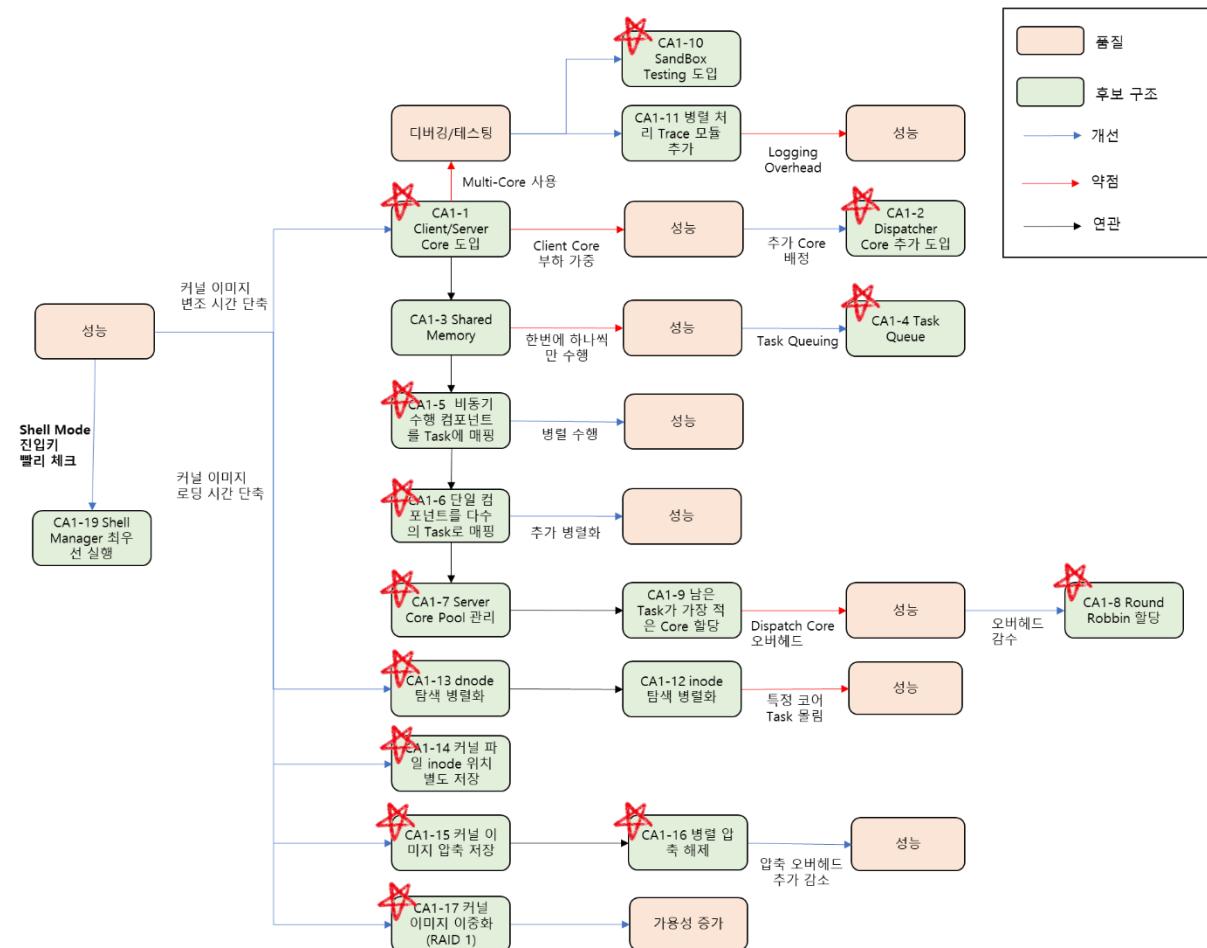


그림 78. QA_01 채택 후보 구조.

E1.1. CA1-1 Client/Server Core 도입 [채택]

E1.2. CA1-2 Dispatcher Core 추가 도입 [채택]

품질 속성	CA1-1 영향	CA1-2 영향
NFR_01 Boot Loader 수행 시간	(+) Boot Loader 수행 시간 중 가장 큰 부분을 차지하는 커널 이미지 로드 시간과 Filesystem Traverse 시간	(+) CA1-1에서 확장된 후보 구조로 Client Core의 부하를 해결 할 수 있다.
QA_01 Boot Loader 수행		

시간	을 병렬화 하여 크게 단축할 수 있는 구조이다. (-) 다수의 Server Core에 Task 요청으로 인한 Client Core 병목에 발생 할 수 있다.	
QA_03 Shell Command 처리 시간	(+) Shell Command 중 처리 시간이 오래 걸리는 부팅, 파일 관련 Command 처리에 적용 가능하며, 해당 시간을 크게 단축할 수 있다. (-) 다수의 Server Core에 Task 요청으로 인한 Client Core 병목에 발생 할 수 있다.	(+) CA1-1에서 확장된 후보 구조로 서비스 Client Core의 부하를 해결할 수 있다.
개발 비용	(-) 순차 처리 구조와 비교하여 병렬화 수행으로 인한, 디버깅과 테스팅이 어렵다.	(-) 병렬화 구조로 인한 디버깅과 테스팅이 어렵다.

[선정 이유] CA1-1은 수행 시간 단축을 위한 핵심 병렬화 구조로, CA1-2를 함께 채택할 경우, 개발 비용 외에 다른 품질 속성에 부정적 영향이 없다고 판단된다. 따라서 CA1-1과 CA1-2를 함께 채택하였다.

[Risk] 병렬화로 인한 디버깅과 테스팅이 어렵다. 이에 따른 완화 방안은 CA1-10/CA1-11에서 검토되었다.

E1.3. CA1-3 코어 간 Shared Memory를 통한 통신 **[미채택]**

E1.4. CA1-4 코어 간 Task Queue를 통한 통신 **[채택]**

품질 속성	CA1-3 영향	CA1-4 영향
NFR_01 Boot Loader 수행 시간	(+) 병렬 처리를 통한 성능 _향_상_	
QA_01 Boot Loader 수행 시간		
QA_03 Shell Command 처리 시간		(++) Client Core가 여러 개의 Task를 Queuing할 수 있어, Server Core가 연속적인 Task 처리가 가능하다.

[선정 이유] CA1-3과 CA1-4 모두 CA1-1과 CA1-2와 연관된 구조로, 병렬 처리를 통한 성능 _향_상_이 가능하다. 추가로, CA1-4의 경우, Server Core에서 연속적인 Task 처리가 가능하여 CA1-3보다 성능 _향_상_이 크다고 판단되어 CA1-4를 채택하였다.

E1.5. CA1-5 Client Core와 비동기 수행 컴포넌트를 Server Core에 할당 [채택]

E1.6. CA1-6 단일 컴포넌트를 다수의 Server Core에 할당 [채택]

품질 속성	CA1-5 영향	CA1-6 영향
NFR_01 Boot Loader 수행 시간 QA_01 Boot Loader 수행 시간 QA_03 Shell Command 처리 시간	(+) 동시 처리를 통한 수행 시간 단축 가능	
		(++) 단일 Task를 Sub-task로 나누어 동시 처리 수행 (-) 특정 Core에 Task가 몰릴 수 있음

[선정 이유] CA1-5와 CA1-6은 모두 Server Core에서 수행할 Task 설계에 관련된 내용이다. 두 후보 구조 모두 함께 적용 가능하며, 동시 처리를 통한 성능 _향_상_이 가능하여, 모두 채택하였다.

[Risk] CA1-6은 단일 Task를 어떻게 분배 할거냐에 따라 특정 Core에 Task가 몰릴 수 있다. 이에 따른 추가 설계 방안은 후보 구조 CA1-7/CA1-8/C1-9에서 검토되었다

E1.7. CA1-7 Server Core Pool 관리 [채택]

품질 속성	CA1-7 영향
NFR_01 Boot Loader 수행 시간 QA_01 Boot Loader 수행 시간 QA_03 Shell Command 처리 시간	(++) 유튜 Server Core의 사용율을 높여, 성능 _향_상_이 가능하다.

[선정 이유] CA1-7은 CA1-6의 Risk를 보완하기 위한 방안으로, Server Core Pool을 구성하여, 병렬화를 극대화하는 방식이다. 따라서 이로 인한 성능 _향_상_ 이점을 매우 커서 본 구조를 채택하였다.

E1.8. CA1-8 Round Robin 할당 [채택]

E1.9. CA1-9 남은 Task가 가장 적은 Core 할당 [미채택]

품질 속성	CA1-8 영향	CA1-9 영향
-------	----------	----------

NFR_01 Boot Loader 수행 시간 QA_01 Boot Loader 수행 시간 QA_03 Shell Command 처리 시간	(+) Dispatch Core의 오버헤드가 작다. (-) Filesystem Traverse시 특정 Core에게 수행 시간이 오래 걸리는 Task가 몰릴 수 있다.	(+) Task 수행 Latency 단축이 가능하다. (--) Task Queue 상태 파악을 위한 Dispatch Core 추가 오버헤드가 필요하다.
--	---	---

[선정 이유] CA1-8과 CA1-9는 CA1-7 Server Core Pool에 Task 배분 설계 방안이다. CPU Core 수가 약 50~100개로 매우 많은 데이터센터 서버에서 CA1-9는 Server Core별 Task Queue 상태를 파악하고, 남은 Task를 기준으로 매번 Sorting하기 위한 Dispatch Core에 오버헤드가 매우 크다. 따라서 비교적 간단하지만, 성능 _향_상_이 큰 CA1-8을 채택하였다.

[Risk] CA1-8은 Filesystem Traverse시 특정 Core에게 수행 시간이 매우 오래 걸리는 Task가 몰릴 수 있다. 따라서 각 Task를 균등 배분하는 것이 중요하며, 이에 대한 완화 방안은 CA1-12/CA1-13에서 다룬다.

E1.10. CA1-10 Sandbox Testing 도입 [채택]

품질 속성	CA1-10 영향
QA_02 디바이스 추가 및 변경 용이성	(+) 디바이스 추가 시 병렬처리 Test 가능
Testability	(++) Test를 위한 모듈 추가로 병렬처리 Test 가능

[선정 이유] CA1-10은 CA1-1의 Risk를 보완하기 위한 방안으로, 병렬화 Test를 위한 모듈을 도입하는 방안이다. 해당 후보 구조는 개발 중에 사용되므로, 실제 사용자가 체감하는 성능에는 영향을 미치지 않아 채택하였다.

E1.11. CA1-11 병렬처리 Trace 모듈 추가 [미채택]

품질 속성	CA1-11 영향
NFR_01 Boot Loader 수행 시간	(-) Log 기록을 위한 오버헤드가 추가된다.
QA_01 Boot Loader 수행 시간	
QA_03 Shell Command 처리 시간	
Debuggability	(+) 병렬처리 도중 Fail 발생 시, 분석 가능

[미선정 이유] CA1-10 Sandbox Testing 도입을 통해, 병렬 처리의 안정성을 충분히 확보한다면, 본 후보 구조의 경우, 성능 감소의 단점만 남게 되어, 채택하지 않았다.

E1.12. CA1-12 Server Core간 inode 기준 탐색 병렬화 [미채택]

E1.13. CA1-13 Server Core간 dnode 기준 탐색 병렬화 [채택]

품질 속성	CA1-12 영향	CA1-13 영향
NFR_01 Boot Loader 수행 시간	(-) 하나의 inode가 가질 수 있는 dnode 수는 1 ~ 약 10	(+) 모든 Server Core에 4KB 크기의 Task 분배 가능함.
QA_01 Boot Loader 수행 시간		(-) Dispatch Core의 Task 분배 오버헤드가 병목이 될 수 있음.
QA_03 Shell Command 처리 시간	역개로 Server Core 간의 Task 균등 분배가 불가능함.	

[선정 이유] CA1-13은 모든 Server Core에 균등한 Task 배분이 가능해, 병렬화를 통한 성능 향상이 크다. 또한, 각 컴포넌트의 수행 시간이 워크로드 (inode당 dnode 수)에 크게 영향을 받지 않아, 시스템 최적화 관점에서 유리하다.

[Risk] Dispatch Core Task 분배 오버헤드가 클 경우, 이를 줄일 수 있는 추가 최적화 방안이 필요하다.

E1.14. CA1-14 커널 이미지 파일 inode 위치 별도 저장 [채택]

품질 속성	CA1-14 영향
NFR_01 Boot Loader 수행 시간	(+) 커널 이미지 접근을 위한 파일 시스템 Traverse 과정을 생략할 수 있다.
QA_01 Boot Loader 수행 시간	
QA_03 Shell Command 처리 시간	

[선정 이유] CA1-14은 inode 위치의 별도 저장을 위해 128B 크기의 추가 저장 공간만 필요하지만, 이로 인한 성능 향상 이점을 매우 커서 본 구조를 채택하였다.

E1.15. CA1-15 커널 이미지 압축 저장 [채택]

E1.16. CA1-16 커널 이미지 병렬 압축 해제 [채택]

품질 속성	CA1-15 영향	CA1-16 영향
NFR_01 Boot Loader 수	(+) 커널 이미지 로드를 위한 스토	(+) 커널 이미지 로드를 위한 스

행 시간 QA_01 Boot Loader 수행 시간 QA_03 Shell Command 처리 시간	리지 접근 시간이 줄어든다. (-) 압축 해제를 위한 추가 프로세스 과정이 필요하다.	토리지 접근 시간이 줄어든다. (+) 압축 해제 오버헤드를 줄일 수 있다.
--	--	--

[선정 이유] CA1-15와 CA1-16을 통한 이미지 압축 저장으로 얻을 수 있는 스토리지 접근 시간 감소는 약 70% 정도로, 압축 해제를 위해 증가되는 시간보다 더 크다. 또한, CA1-16을 통해, 압축 해제 시간을 추가적으로 줄일 수 있어, 모두 채택하였다.

[Risk] 병렬화 압축 해제가 가능하도록, 커널 설치 시 커널 이미지를 저장하여야 한다. 만약 그렇지 않다면, 추가적인 커널 이미지 재 압축 설치가 필요하다.

E1.17. CA1-17 커널 이미지 이중화 (RAID 1) [채택]

품질 속성	CA1-17 영향
NFR_01 Boot Loader 수행 시간 QA_01 Boot Loader 수행 시간 QA_03 Shell Command 처리 시간	(+) 2개의 부팅 드라이브에서 병렬적으로 커널 이미지 로드가 가능하다.
Availability	(+) 하나의 부팅 드라이브가 고장 나더라도 부팅 가능하다.
저장 비용	(-) 동일 커널 이미지 저장을 위한 추가 스토리지 장치가 필요하다.

[선정 이유] 데이터센터 서버의 경우, Availability _향_상_을 위해 부팅 드라이브 2개를 RAID 1으로 구성하고 있다. 따라서 본 후보 구조의 약점이 사라지게 된다.

E1.18. CA1-18 Shell Manager 최우선 수행 [채택]

품질 속성	CA1-18 영향
NFR_01 Boot Loader 수행 시간 QA_01 Boot Loader 수행 시간	(+) Shell Mode 진입 가능 시점을 최대한 앞당겨, 전체 부팅 시간 단축 가능하다.

[선정 이유] CA1-18은 시간이 가장 오래 걸리는 사용자 Shell Mode 진입 대기를 위해, Shell Mode 진입 가능 시점을 최대한 앞당겨, 전체 부팅 시간을 단축 가능하다. 또한, Shell Manager 우선 수행으로 인해, 다른 성능에 영향을 거의 미치지 않아, 본 후보 구조를 채택하였다.

E2. QA_02 디바이스 추가 및 변경 용이성

디바이스 추가 및 변경 용이성 관련 후보 구조 선정 결과를 요약하면 아래 그림과 같다.

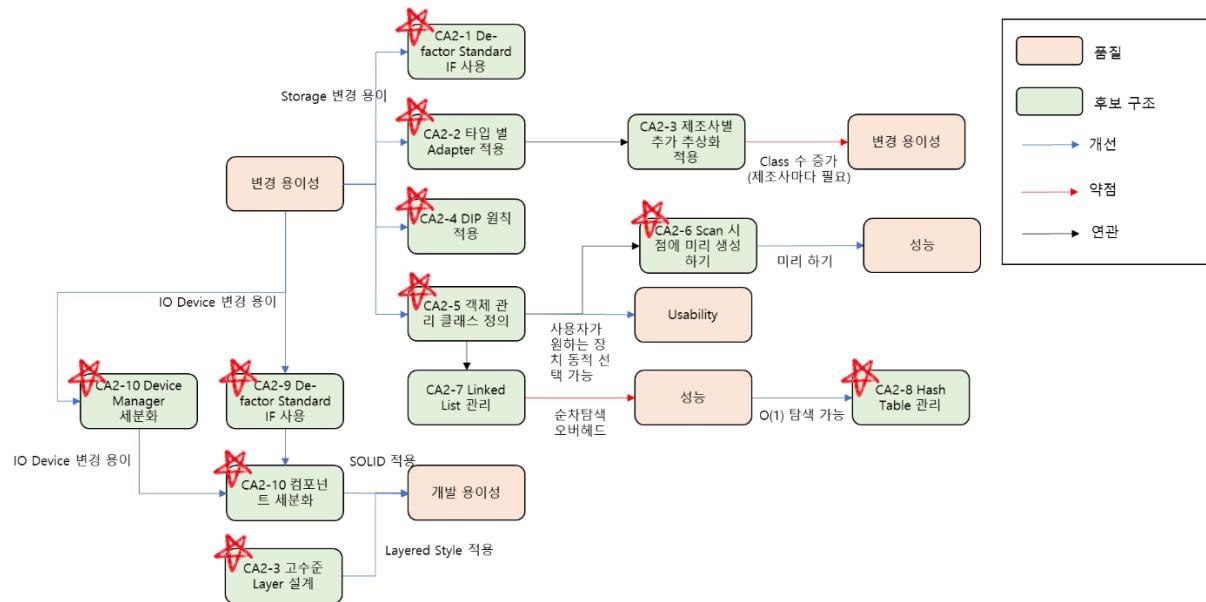


그림 79. QA_02 채택 후보 구조.

E2.1. CA2-1 Storage Device를 위해 De-factor Standard 인터페이스 사용 [채택]

품질 속성	CA2-1 영향
QA_02 디바이스 추가 및 변경 용이성	(+) Storage 인터페이스를 통해, 다양성 확보 및 변경 용이성 강화가 가능함.

[선정 이유] CA2-1 구조는 Storage Device에 대한 추가 및 변경 용이성을 달성하는데 가장 기본적인 시작이 되는 구조로, 이를 선정하였다.

E2.2. CA2-2 스토리지 타입 별 Adapter 적용 [채택]

E2.3. CA2-3 제조사별 추가 추상화 [미채택]

E2.4. CA2-4 DIP 원칙 적용 [채택]

품질 속성	CA2-2 영향	CA2-3 영향	CA2-4 영향
-------	----------	----------	----------

QA_02 디바이스 추가 및 변경 용이성	(+) 추상화로 인한 확장성과 재사용성 높아짐. (-) 스토리지 타입 별 Adaptor가 필요.	(--) 제조사마다 추가적인 Adaptor가 더 필요하다. .	(+) 코드 구조가 가장 간단하고, 스토리지 변경 및 추가에 따른 코드 변경이 가장 적다.
NFR_01 Boot Loader 수행 시간 QA_01 Boot Loader 수행 시간		(0) 제조사 특화 인터페이스로 인한 추가적인 성능 향상 이점을 기대하기 어렵다	
개발 비용			(-) DIP 적용을 위한 Device Driver의 수정이 1회 필요하다.

[선정 이유] CA2-5은 Device Driver를 변경해야 한다는 큰 단점이 있으나, 다행히도 이미 De-facto Standard화되어 각 제조사마다 인터페이스가 통일된 Device Driver를 제공하고 있다. 따라서 CA2-5의 단점이 이미 해결되었으며, 해당 구조를 채택하였다. 또한, 공통 인터페이스를 제공하지 않는 신규 Device Driver의 경우, CA2-3과 같이 별도의 Adapter가 필요하다. 따라서 CA2-3도 함께 채택하였다.

E2.5. CA2-5 Storage Device 객체 관리 클래스 정의 [채택]

품질 속성	CA2-5 영향
QA_02 디바이스 추가 및 변경 용이성	(+) 사용자의 부팅 스토리지 변경 요청에 코드 변화없이 대응 가능
Usability	(++) 사용자는 Shell CMD를 통해, 스토리지를 동적으로 선택 가능

[선정 이유] CA2-5는 QA_02 _향_상_과 더불어, 사용자 편의성을 증가시킬 수 있어 채택하였다.

E2.6. CA2-6 Scan 시점에 미리 Storage Device 객체 생성하기 [채택]

품질 속성	CA2-6 영향
-------	----------

QA_03 Shell Command 처리 시간	(+) 사용자의 Storage Device 변경 요청 시점이 아닌 Storage Device Scan 시점에 미리 객체들을 생성하여, Shell CMD의 수행 성능이 향상된다.
저장 비용	(-) 사용자가 어떠한 Storage Device로의 변경을 원할지 알 수 없기 때문에, 모든 Storage Device에 대한 객체를 미리 생성하여야 한다.

[선정 이유] CA2-6은 QA_03 개선에 도움이 되는 구조로 이를 채택하였다. 메모리 공간 추가 오버헤드의 경우, 데이터센터 서버에 장착되는 Service Device 수가 10개 ~ 40개임을 고려할 때, 메모리 공간 오버헤드는 거의 없을 것으로 판단하였다.

E2.7. CA2-7 Linked List를 통한 Device 객체 관리 [미채택]

E2.8. CA2-8 Hash Table을 통한 Device 객체 관리 [채택]

품질 속성	CA2-7 영향	CA2-8 영향
QA_02 디바이스 추가 및 변경 용이성	(+) Storage Device 객체의 동적 관리가 가능한 구조이다.	
NFR_01 Boot Loader 수행 시간 QA_01 Boot Loader 수행 시간 QA_03 Shell Command 처리 시간	(-) 원하는 Device에 대한 순차 탐색이 필요하다.	(+) Device 탐색 복잡도는 O(1)이다. (-) Device 객체 삽입 시 Hash 키 생성에 대한 추가 프로세싱 시간이 필요하다.

[선정 이유] 두 구조 모두 서버에 장착된 Storage Device 목록을 동적으로 관리 가능하다. 다만, CA2-8의 경우, 객체 탐색 성능에 장점이 있어 해당 구조를 채택하였다.

E2.9. CA2-9 IO Device를 위해 De-factor Standard 인터페이스 사용 [채택]

품질 속성	CA2-9 영향
QA_02 디바이스 추가 및 변경 용이성	(+) De-factor Standard Serial 인터페이스를 통해, 다양성 확보 및 변경 용이성 강화가 가능함.

[선정 이유] CA2-1은 Serial 인터페이스를 통한 IO Device 대한 추가 및 변경 용이성을 달성하는데 필요한 구조로, 이를 선정하였다.

E2.10. CA2-10 Device Manager 세분화 [채택]

품질 속성	CA2-10 영향
QA_02 디바이스 추가 및 변경 용이성	(+) Storage Manager와 IO Device Manager를 S_R_P를 적용하여 두 개의 모듈로 분리. 이를 통해, 변경 용이성 추가 _향_상_.

[선정 이유] CA2-10은 S_R_P를 적용하여 역할과 인터페이스가 서로 다른 두 개의 모듈을 각자 분리함으로써, 변경 용이성을 _향_상_된다. 따라서, 해당 구조를 선정하였다.

E2.11. CA2-11 컴포넌트 세분화 [채택]

품질 속성	CA2-11 영향
QA_02 디바이스 추가 및 변경 용이성	(+) S_R_P, DIP 적용을 통한 변경 용이성 _향_상_.
개발 용이성	(+) Device Manager 참조 컴포넌트에 대한 변경 용이성 _향_상_.

[선정 이유] CA2-11은 Layered Style 적용을 위해, 도메인 모델에서 도출된 컴포넌트들에 대해 SOLID 원칙을 적용하였다. 이를 통해, QA_02 개선 및 개발 용이성 개선을 얻을 수 있어, 본 구조를 선정하였다.

E2.12. CA2-12 고수준 Layer 설계 [채택]

품질 속성	CA2-12 영향
개발 용이성	(+) Layered Style 적용을 통한 개발 용이성 _향_상_.

[선정 이유] CA2-12는 QA_02와 직접 연관은 없지만, Layered Style 적용을 통해, 개발 용이성을 개선할 수 있어, 본 구조를 선정하였다.

E3. QA_03 Shell Command 처리 시간

Shell Command 관련 후보 구조 선정 결과를 요약하면 아래 그림과 같다.

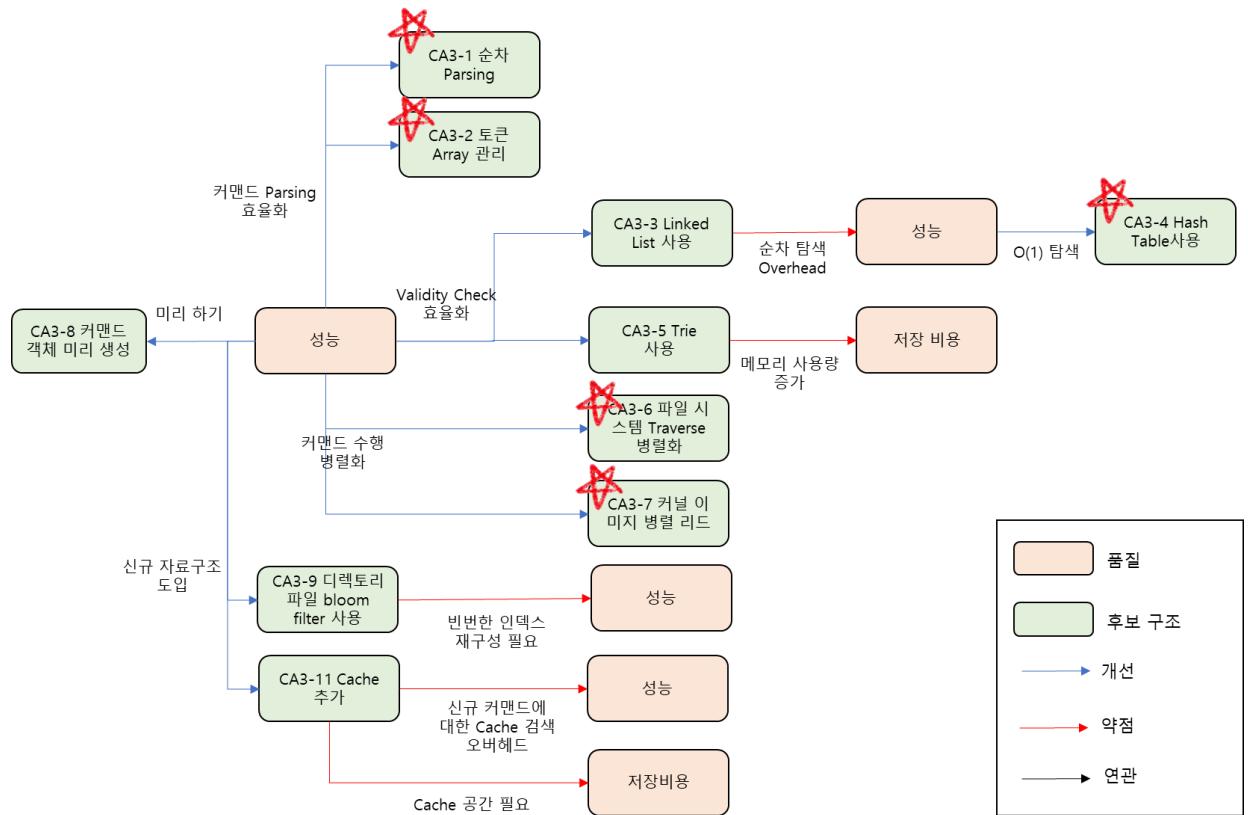


그림 80. QA_03 채택 후보 구조.

E3.1. CA3-1 커맨드 순차 Parsing [채택]

품질 속성	CA3-1 영향
QA_03 Shell Command 처리 시간	(+) Tokenize를 위한 기본 방안으로 사용자가 입력한 커맨드 문자열을 순차 탐색을 통해 Parsing하는 방안

[선정 이유] Shell Command는 커맨드별 토큰이 최대 3개이며, 문자열 길이도 평균 10개 이하이다. 따라서 순차 탐색을 빠르게 수행할 수 있어, 본 구조를 선택하였다.

E3.2. CA3-2 Array를 통한 토큰 관리 [채택]

품질 속성	CA3-2 영향
QA_03 Shell Command 처리 시간	(+) Tokenize 결과 생성된 토큰들을 Linked List로 관리 방 안.

[선정 이유] Shell Command는 최대 3개의 토큰을 가진다. 따라서, CA3-2는 적은 수의 토큰을 빠
르게 관리할 수 있어, 선택하였다.

E3.3. CA3-3 커맨드/Arg 리스트 관리를 위한 Linked List 사용 [미채택]

E3.4. CA3-4 커맨드/Arg 리스트 관리를 위한 Hash Table 사용 [채택]

E3.5. CA3-5 커맨드/Arg 리스트 관리를 위한 Trie 사용 [미채택]

품질 속성	CA3-3 영향	CA3-4 영향	CA3-5 영향
QA_03 Shell Command 처리 시간	(+) 커맨드 유효성을 빠르게 체크할 수 있다. (-) 순차 탐색 오버헤드	(+) O(1) 탐색 가능	(+) O(height) 탐색 가능
저장 공간			(-) 커맨드 문자열 길 이가 길어질수록 기하 급수적으로 메모리 공 간이 많이 필요하다.

[선정 이유] CA3-4는 CA3-5와 유사한 수준의 Shell Command 처리 시간이 기대되며, 필요한 저장
공간이 적어, 이를 채택하였다.

E3.6. CA3-6 파일 시스템 Traverse 병렬화 [채택]

E3.7. CA3-7 커널 이미지 로드 병렬화 [채택]

품질 속성	CA3-6 영향	CA3-7 영향
NFR_01 Boot Loader 수행 시간	(++) Multi-Core 병렬화를 통한 수행 시간을 크게 단축할 수 있음	
QA_01 Boot Loader 수행 시간		

QA_03 Shell Command 처리 시간	
---------------------------	--

[선정 이유] CA3-6과 CA3-7은 병렬화를 통해 수행 시간을 크게 단축할 수 있으며, NFR_01, QA_01 품질 _향_상_도 가능함으로 이를 채택하였다.

E3.8. CA3-8 커맨드 객체 미리 생성 [채택]

품질 속성	CA3-8 영향
QA_03 Shell Command 처리 시간	(+) Shell Command 수행을 위한 객체를 수행 시점이 아닌, 커맨드 리스트 구축 시점에 미리 생성하여, Shell Command의 수행 시간 단축 가능
리소스 사용	(-) 모든 커맨드 객체를 미리 생성 필요하여, 객체를 저장할 추가적인 메모리 공간이 필요.

[선정 이유] CA3-8은 Shell Command 처리 시간을 단축시킬 수 있어, 이를 채택하였다. CA3-8의 약점인 추가 메모리 공간 필요의 경우다, Shell Command는 총 17개로, 해당 객체를 미리 저장하기 위한 오버헤드는 거의 없을 것으로 판단하였다.

E3.9. CA3-9 디렉토리 파일 Bloom Filter 구축 [미채택]

품질 속성	CA3-9 영향
NFR_01 Boot Loader 수행 시간	(+) 디렉토리 안의 파일 존재 여부 탐색 시간을 크게 단축할 수 있음.
QA_01 Boot Loader 수행 시간	(-) find 커맨드에서만 큰 성능 _향_상_을 기대할 수 있다.
QA_03 Shell Command 처리 시간	(-) 디렉토리 내의 파일이 추가/삭제될 경우, Bloom Filter를 재구성하여야 한다.
리소스 사용	(-) Bloom Filter 적재를 위한 추가 DRAM 공간 필요함.
개발 비용	(-) 복잡도가 높은 Bloom Filter 추가 개발을 하여야 한다.

[미선정 이유] CA3-9 구조는 탐색 성능 _향_상_을 기대할 수 있으나, 적용할 수 있는 커맨드가 매우 제한적이고, 빈번한 재구성을 수행하여야 한다. 따라서 본 구조를 선정하지 않았다.

E3.10. CA3-10 Cache 추가 [미채택]

품질 속성	CA3-10 영향
QA_03 Shell Command 처리 시간	(+) 반복 사용하는 커맨드에 대한 수행 시간을 단축 가능하다. (-) 신규 커맨드의 경우, Cache 검색 오버헤드가 추가된다.
리소스 사용	(-) Caching을 위한 추가 DRAM 공간 필요

[미선정 이유] CA3-10 구조는 Caching을 커맨드 탐색 시간 및 Filesystem Traverse 성능 향상을 기대할 수 있다. 그러나, Boot Loader의 경우, Shell Mode에서 사용자가 특정 커맨드를 반복하여 사용하는 경우가 드물어, 오히려, Cache 검색 오버헤드 추가로 인한 성능 저하가 발생할 것으로 판단되어 채택하지 않았다.

E4. QA_04 커널 이미지 로딩 가용성

커널 이미지 로딩 가용성 관련 후보 구조 선정 결과를 요약하면 아래 그림과 같다.

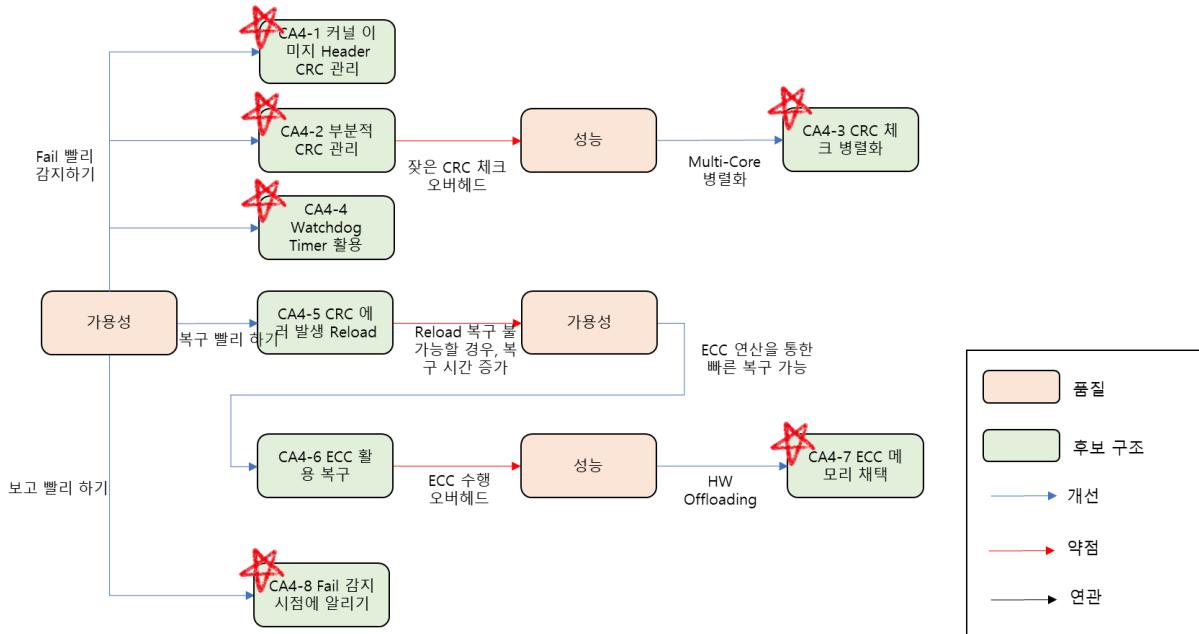


그림 81. QA_04 채택 후보 구조.

E4.1. CA4-1 커널 이미지 Header CRC 관리 [채택]

품질 속성	CA4-1 영향
QA_04 커널 이미지 로딩 가용성	(+) 커널 이미지 Header CRC 추가를 통해, Header 장애 발생을 빨리 감지 가능
저장 비용	(0) CRC 추가 저장을 위한 공간 필요 (4B)

[선정 이유] CA4-1은 Header CRC 추가를 통해 장애 발생을 빨리 감지 가능하다. 또한, 128B Header에 대해 4B 공간 추가되므로, 저장 공간 오버헤드가 크지 않다고 판단하여 선정하였다.

E4.2. CA4-2 커널 이미지 부분적 CRC 관리 [채택]

E4.3. CA4-3 CRC 체크 병렬화 관리 [채택]

품질 속성	CA4-2 영향	CA4-3 영향
-------	----------	----------

QA_04 커널 이미지 로딩 가용성	(+) 전체 커널 이미지를 로딩 하지 않고도, 커널 이미지의 특정 부분에 발생한 Fail을 빨리 감지할 수 있음	
NFR_01 Boot Loader 수행 시간	(-) 커널 이미지 로딩 중 잦은 CRC 체크가 필요	(+) Multi-Core 병렬화를 통한 CRC 체크 수행 시간 단축
QA_01 Boot Loader 수행 시간		
QA_03 Shell Command 처리 시간		
저장 비용	(0) 부분 이미지마다 4B CRC 공간 필요	

[선정 이유] CA4-2와 CA4-3은 커널 이미지에 대한 부분적 CRC를 추가함으로써, 전체 커널 이미지를 로딩 하지 않고도, 특정 부분에 발생한 장애를 빨리 감지할 수 있다. 또한, CA4-3은 CA4-2의 성능 저하 단점을 개선할 수 있어, 두 후보 구조 모두를 선정하였다.

E4.4. CA4-4 Watchdog Timer 활용 [채택]

품질 속성	CA4-4 영향
QA_04 커널 이미지 로딩 가용성	(+) 예상하지 못한 Hang 발생 시 빠른 장애 감지가 가능

[선정 이유] CA4-4는 예상치 못한 Hang 발생 시 빠른 장애 감지가 가능하여, 이를 선정하였다.

E4.5. CA4-5 CRC 에러 발생 시 Reload [미채택]

품질 속성	CA4-5 영향
QA_04 커널 이미지 로딩 가용성	(+) CRC 에러 발생 시, Storage에서 재로드 하는 방식으로, 사용자의 대응없이 복구 가능하여, 복구 시간을 단축할 수 있음. (-) Reload로 복구 불가능한 장애일 경우, 오히려 복구 시간이 늘어남.
NFR_01 Boot Loader 수행 시간	(-) Reload로 인한 커널 이미지 로드 시간 증가
QA_01 Boot Loader 수행 시간	
QA_03 Shell Command 처리 시간	

[미선정 이유] CA4-5는 Reload를 통해, Storage에서 DRAM으로 커널 이미지를 읽어오는 과정에서 발생한 장애에 대해 사용자 대응 없이 복구 가능하지만, Reload를 통해 복구 불가능한 장애일 경우, 오히려 복구 시간이 늘어나는 Risk가 있어, 선정하지 않았다.

[Risk] Reload로 복구 불가능한 장애일 경우, 오히려 사용자 개입 시간을 늦출 수 있다. 이를 완화하는 방안은 CA4-8에서 다룬다.

E4.6. CA4-6 ECC 활용 커널 이미지 복구 **[미채택]**

E4.7. CA4-7 ECC 메모리 채택 **[채택]**

품질 속성	CA4-6 영향	CA4-7 영향
QA_04 커널 이미지 로딩 가용성	(+) Storage에서 다시 로드 하는 것보다 빠르게 장애를 복구할 수 있다.	
NFR_01 Boot Loader 수행 시간 QA_01 Boot Loader 수행 시간 QA_03 Shell Command 처리 시간	(-) ECC 수행 시간 필요	(+) ECC 제공 DRAM을 통해, 해당 오버헤드 제거 가능
저장 비용		(-) 가격이 비싼 ECC 제공 DRAM 필요

[선정 이유] 데이터센터 서버에서 사용하는 서버급 메모리 (ex. ECC Registered DRAM)은 ECC 기능을 이미 제공하고 있다. 따라서 본 후보 구조의 약점이 사라지게 되어, CA4-7을 선정하였다.

E4.8. CA4-8 Fail 감지 시점에 알리기 **[채택]**

품질 속성	CA4-8 영향
QA_04 커널 이미지 로딩 가용성	(+) 미리 사용자에게 알림으로써, 복구 불가능 장애 발생 시 사용자 인지 시점을 앞당길 수 있게 된다.

[선정 이유] CA4-8은 사용자 개입이 필요한 복구의 경우, 개입 시간 단축을 위해 사용자에게 빨리 알릴 수 있으며, 이로 인한 Risk가 없어 선정하였다.

E5. QA_05 Shell Command 추가 및 확장 용이성

Shell Command 추가 확장 용이성 관련 후보 구조 선정 결과를 요약하면 아래 그림과 같다.

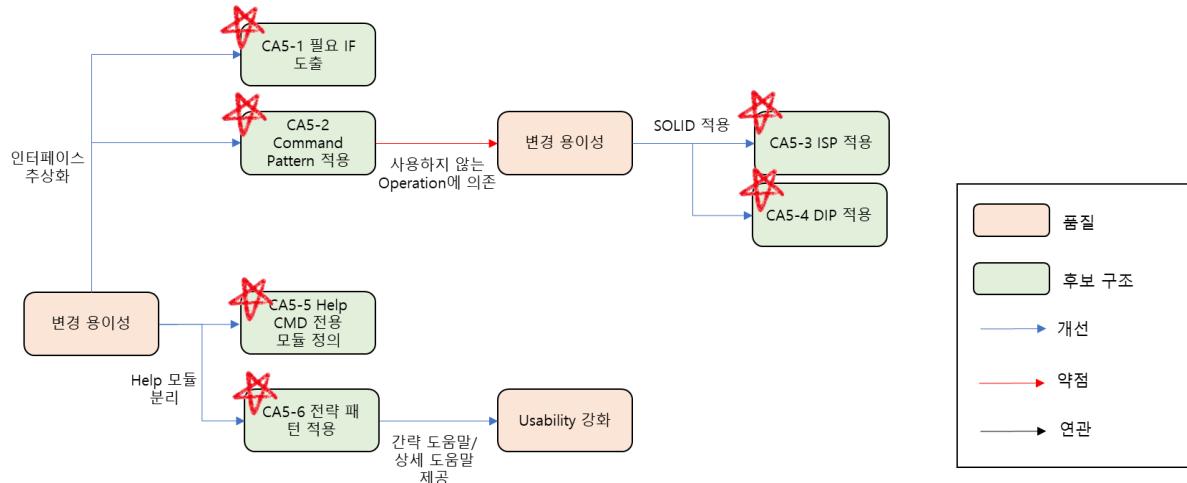


그림 82. QA_05 채택 후보 구조.

E5.1. CA5-1 Shell Command 필요 인터페이스 도출 [채택]

품질 속성	CA5-1 영향
QA_05 Shell Command 추가 및 확장 용이성	(+) Command 인터페이스를 통해, 다형성 확보 및 변경 용이성 강화가 가능함.

[선정 이유] CA5-1은 Shell Command 추가 및 확장 용이성을 달성하는데 가장 기본적인 시작이 되는 구조로, 이를 선정하였다.

E5.2. CA5-2 Command Pattern 적용 [채택]

품질 속성	CA5-2 영향
QA_05 Shell Command 추가 및 확장 용이성	(+) 커맨드 제공 클래스에 대한 의존성 제거 가능하여, 커맨드 확장성이 높아짐. (-) 특정 클라이언트 컴포넌트는 사용하지 않는 Operation에 의존한다.

[선정 이유] CA5-2은 Shell Command 추가 및 확장 용이성을 달성할 수 있는 구조로, 이를 선정하였다.

[Risk] 클라이언트 컴포넌트들은 자신이 사용하지 않는 Operation에 의존하게 된다. 이로 인해, QA_05을 약화시킨다. 이에 대한 보완은 CA5-3에서 다룬다.

E5.3. CA5-3 Command 인터페이스에 ISP 적용 [채택]

E5.4. CA5-4 DIP 적용 [채택]

품질 속성	CA5-3 영향	CA5-4 영향
QA_05 Shell Command 추가 및 확장 용이성	(+) 큰 덩어리의 인터페이스를 작은 단위의 인터페이스로 분리함으로써, 변경 용이성이 _향_상_된다.	(+) 상위 계층인 Shell Manager에 의존성을 역전시킴으로써, 신규 커맨드 추가 및 확장 시에도 상위 컴포넌트의 변경되지 않는다.

[선정 이유] CA5-3, CA5-4는 SOLID 원칙 중 각각 ISP, DIP를 적용하여 QA_05를 개선시킨다. 특히, CA5-3은 CA5-2의 단점을 보완 가능하다.

E5.5. Help CMD 전용 모듈 정의 [채택]

품질 속성	CA5-5 영향
QA_05 Shell Command 추가 및 확장 용이성	(+) Concrete Command Class가 공통적으로 제공해야 할 도움말 출력 모듈을 분리함으로써, 변경 용이성을 _향_상_된다.

[선정 이유] CA5-5는 S_R_P를 적용하여 Concrete Command Class의 공통부인 도움말 출력 모듈을 분리함으로써, 변경 용이성을 _향_상_시킨다.

E5.6. 도움말 출력 전략 패턴 적용 [채택]

품질 속성	CA5-6 영향
QA_05 Shell Command 추가 및 확장 용이성	(+) 사용자 요청에 따라 요약 도움말과 상세 도움말 출력을 선택할 수 있게 전략 패턴을 적용함으로써, 변경 용이성을 _향_상_됨.
Usability _향_상_	(+) 사용자는 필요한 형태의 도움말 요청이 가능해짐.

[선정 이유] CA5-6는 전략 패턴을 통해, 도움말 기능의 확장 용이성을 _향_상_시킬 수 있어, 이를 선정하였다.

F. 최종 구조 설계

F1. 시스템 동작/실행 최종 구조 설계 경위

본 장에서는 시스템 동작/실행 측면에서 최종 구조를 설계 경위를 설명한다. 아래 항목들은 채택된 후보 구조 중 시스템 동작/실행 측면과 관련된 항목들로, 이들을 반영하여 최종 Deployment View를 설계하였다.

HW 컴포넌트	설계 사항	품질 속성	후보구조
Core	성능 _향_상_을 위한 Multi-Core 병렬화	성능	CA1-1
	Dispatcher Core 추가	성능	CA1-2
	비동기 수행 컴포넌트 Core 채택 (Shell Core, Log Core, IO Device Core)	성능	CA1-5
	Server Core Pool 관리	성능	CA1-7
	Filesystem Traverse 병렬화	성능	CA1-6
	Kernel Image Load 병렬화	성능	CA1-6
	Decompressor 병렬화	성능	CA1-16
	CRC Checker 병렬화	성능	CA4-4
	Watchdog Timer 채택	가용성	CA4-4
DRAM	ECC DRAM 채택	가용성 성능	CA4-7
	Task Queue를 통한 코어간 통신	성능	CA1-4
	Storage 객체 관리	사용성	CA2-5
	커맨드 객체 미리 생성	성능	CA3-8
Storage	커널 이미지 압축 채택	성능	CA1-15
	커널 이미지 이중화 (RAID1) 채택	성능 가용성	CA1-17
	커널 이미지 Header CRC 채택	가용성	CA4-1
	커널 부문 이미지 CRC 채택	가용성	CA4-2
IO Device	IO Device Manager 모듈 분리	변경 용이성	CA2-10
	모듈 공통화	변경 용이성	CA2-11

아래 그림은 본 시스템의 최종 구조에 대한 Deployment View 및 채택된 후보 구조를 나타낸다.

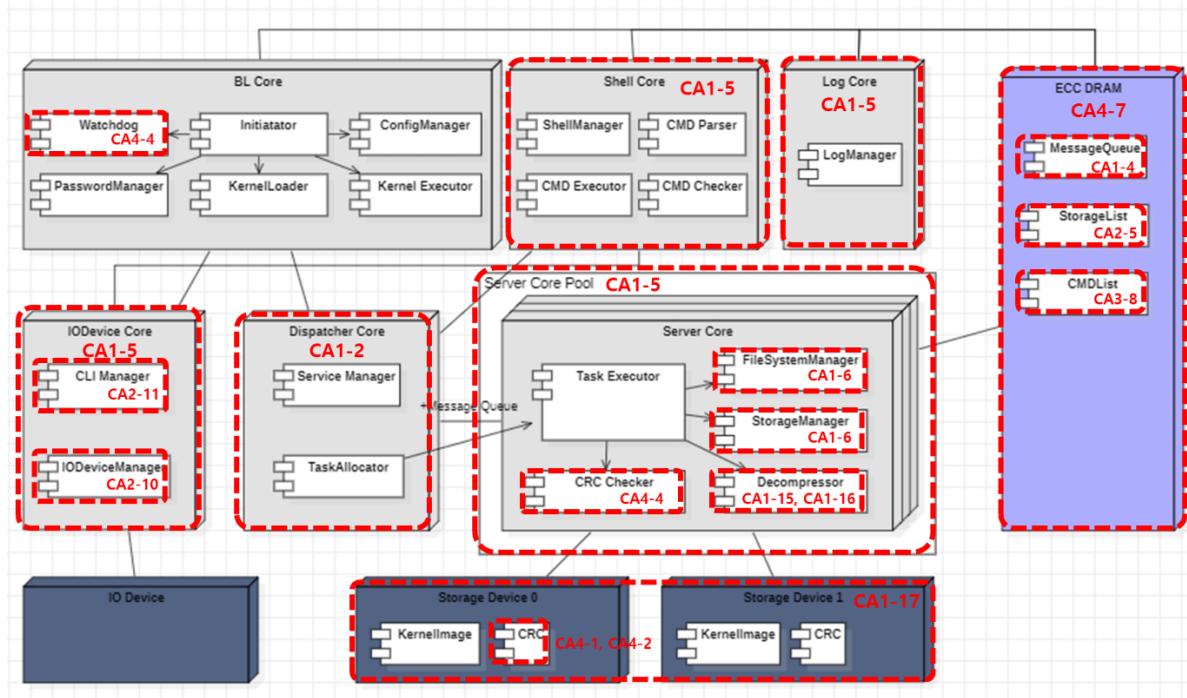


그림 83. Deployment View.

F1.1. CPU 관련 최종 구조

1. BL Core: Boot Loader 핵심 서비스 로직들을 처리하는 Core이며, Dispatcher Core에게 서비스를 요청하는 Client Core.
2. Shell Core: Shell Mode 지원 및 사용자의 Shell Command 입력 처리를 위한 Core. CA1-5에 의해 별도 코어로서 수행이 결정됨.
3. Log Core: Log Message 출력을 위한 Core. CA1-5에 의해 별도 코어로서 수행이 결정됨.
4. IO Device Core: IO Device 관리 전담을 위한 Core. CA1-5에 의해 별도 코어로서 수행이 결정됨.
5. Dispatcher Core: Server Core에서 수행할 서비스들을 관리하고, Client Core와 Server Core 간의 채널을 연결해주는 Core. Client Core 부하 감소를 위해 CA1-2에 의해 해당 코어를 추가하기로 결정됨.
6. Server Core Pool: 동일 Task를 병렬화 수행하는 Core들의 집합. Server Core들 간의 Task

균등 분배를 위해, Server Core Pool을 운영하기로 CA1-7에 의해 결정. Server Core를 통해 수행할 Task는 CA1-6, CA1-16, CA4-4에 의해 결정됨.

F1.2. DRAM 관련 최종 구조

1. ECC DRAM: ECC를 위한 가용성 _향_상_과 ECC 오버헤드 감소를 위해 CA4-7에 의해 채택됨.
2. Message Queue: Core간의 통신을 위한 자료구조. Queuing을 통한 연속 Task 처리를 위해 CA1-4에 의해 결정됨.
3. Device List: 서버에 장착된 Storage 객체들의 리스트. CA2-5에 의해 생성하기로 결정됨.
4. 커맨드 List: Shell CMD 객체들의 리스트. Validity Check를 위해 CA3-8에 의해 생성하기로 결정됨.

F1.3. Storage 관련 최종 구조

1. Kernel Image: 성능 _향_상_을 위해 읽어올 이미지를 압축하여 저장. CA1-15에 의해 결정됨.
2. Kernel Image 이중화: RAID1을 적용하여 성능 및 가용성 _향_상_을 위해 이중화. CA1-17에 의해 결정됨.
3. CRC: 커널 이미지 로딩 시 장애 발생 대응을 위한 용도. CA4-1, CA4-2에 의해 추가하기로 결정됨.

F1.4. IO Device 관련 최종 구조

1. IO Device Manager: IO Device 관리를 전담하는 컴포넌트. CA2-10에 의해 분리하기로 결정됨.
2. CLI Manager: 사용자 입력 값 관리를 전담하는 컴포넌트. CA2-11에 의해 분리하기로 결정됨.

F2. 개발 측면 최종 구조 설계 경위

본 장에서는 개발 측면에서의 최종 구조를 설계 경위를 설명한다. 아래 항목들은 채택된 후보 구조 중 개발 측면과 관련된 항목들로, 이들을 반영하여 최종 Module View를 설계하였다.

관련 레이어/모듈	설계 사항	품질 속성	후보구조
Storage Manager	스토리지 인터페이스 정의	변경 용이성	CA2-1
	스토리지 타입별 Adapter 적용	변경 용이성	CA2-2
	DIP 적용	변경 용이성	CA2-4
	스토리지 객체 관리 클래스 추가	사용성	CA2-5
IODevice Manager	IO Device 인터페이스 정의	변경 용이성	CA2-9
CLI Manager	Password Mgr/Shell Mgr 공통 모듈 추출	변경 용이성	CA2-11
Config Manager	변경 빈번 모듈 분리 (Read/Setting)	변경 용이성	CA2-11
Core Manager	Dispatcher Core 모듈 (Service Mgmt, Server Allocator)	성능	CA1-2
	Core Pool 관리 모듈	성능	CA1-7
	Core간 통신을 위한 Task Queue 모듈	성능	CA1-4
	Round Robin Task 할당 정책	성능	CA1-8
	dnode 기준 할당 정책	성능	CA1-13
FS Manager	커널 이미지에 압축 해제 모듈	성능	CA1-16
	RAID1 지원을 위한 모듈 추가	성능 가용성	CA1-17
	커널 이미지 장애 발생 대응을 위한 CRC 모듈 추가	가용성	CA4-1 CA4-2 CA4-3
Sandbox	병렬 테스트를 위한 테스트 모듈	Debuggability	CA1-10
Shell Manager	Shell Command 인터페이스 정의	변경 용이성	CA5-1
	Command Pattern 적용	변경 용이성	CA5-2
	S_R_P, DIP 적용	변경 용이성	CA5-3 CA5-4
	도움말 출력 공통 모듈 분리 (Helper)	변경 용이성	CA5-5
	요약 도움말/상세 도움말 출력 제공을 위한 전략 패턴 적용	사용성	CA5-6
Device Layer	Storage와 IO Device 관리 Layer 추가	변경 용이성	CA2-10
Core Layer	Device Layer 모듈들과 "allowed to use" 관계에	개발 용이성	CA2-11

	속하는 Layer 추가		CA2-12
Extension Layer	Core Layer 모듈들을 활용한 확장된 기능을 제공하는 Layer 추가	개발 용이성	CA2-12
Business Layer	최상위 Layer로 Boot Loader Business Logic을 제공하는 Layer 추가	개발 용이성	CA2-12

아래 그림은 본 시스템의 최종 구조에 대한 Deployment View 및 채택된 후보 구조를 나타낸다.

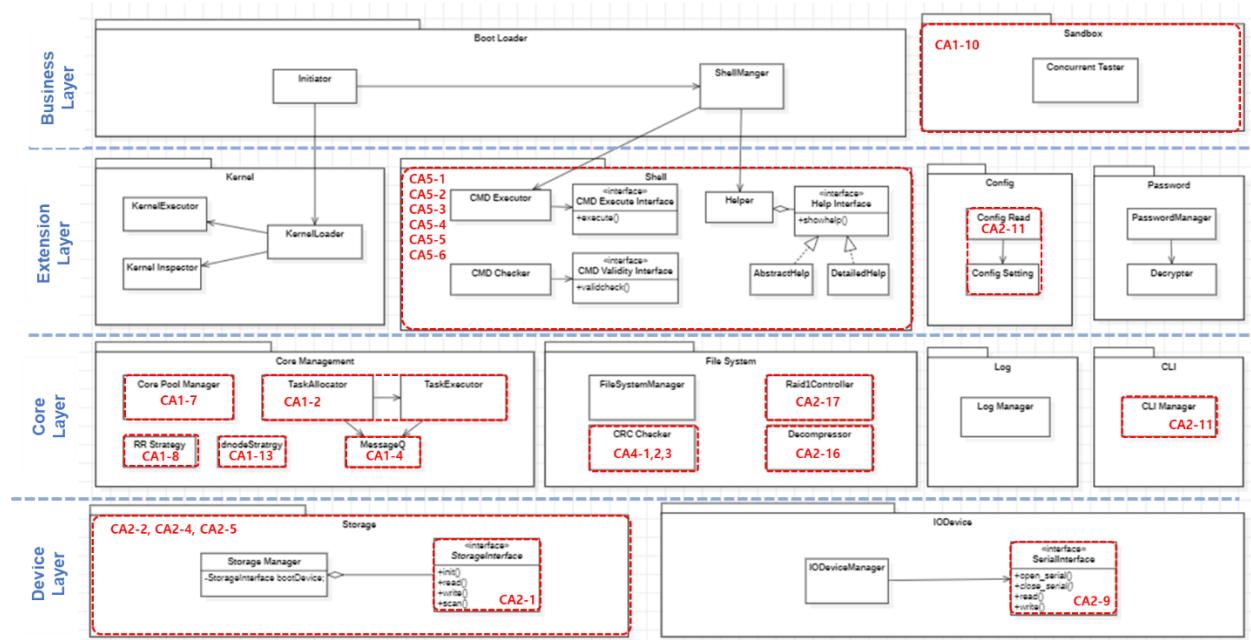


그림 84. Module View.

F2.1. Device Layer

가장 하단 위치의 Layer로 경계인 Storage와 IO Device 접근에 필요한 인터페이스를 제공한다. 해당 Layer에는 Storage 모듈과 IO Device 모듈이 위치하며, 이는 CA2-10에 의해 결정되었다.

F2.2. Core Layer

Device Layer에 속하는 모듈들을 직접적으로 참조 ("allowed to use")하는 모듈들이 속하는 Layer로, 상위 Layer가 필요한 Core Manager (병렬처리), Filesystem Manager, Log, CLI Manager (사용자 커맨드 처리) 관련 기능을 제공하는 모듈들이 위치한다. 해당 Layer는 CA2-12에 의해 결정되었다.

- Core Manager: Multi-Core 병렬화를 위해 필요한 기능 및 인터페이스를 제공. Core Manager에 대한 상세 설계는 CA1-2, CA1-7, CA1-4, CA1-8, CA1-13, CA1-16에 의해 결정되었음.
- Filesystem 모듈: Filesystem 탐색 및 접근을 위한 기능 및 인터페이스 제공. 커널 이미지 압축 해제 (CA1-16), RAID1 (CA1-17), CRC 모듈 (CA4-1, CA4-2)이 추가되었음.
- Log 모듈: Log 출력에 관한 기능을 제공. IO Device 모듈을 직접 접근함.
- CLI 모듈: 사용자 커맨드 처리를 담당하는 모듈로 CA2-11에 의해 공통 모듈 추출이 결정되었음.

F2.3. Extension Layer

Core Layer 모듈들을 제공하는 기능과 인터페이스를 통해, 확장된 기능을 제공하는 Layer로, CA2-12에 의해 결정되었다.

- Kernel 모듈: Kernel 로딩 및 Kernel 수행을 담당하는 모듈.
- Shell 모듈: Shell Command Parsing, Execution, 도움말 출력을 담당하는 모듈. 상세 설계는 CA5-1~CA5-6에 의해 결정되었음.

F2.4. Business Layer

Business Logic을 제공하는 최상위 layer로, 마찬가지로 CA2-12에 의해 결정되었다.

- Initiator: Boot Loader 서비스 위해 필요한 Sequence를 명세하고, Extension Layer에 속하는 모듈을 호출하는 모듈.
- Shell Manager: Shell Mode 제공을 위해 필요한 Sequence를 명세하고 있는 모듈.

F3. 최종 구조의 단점 및 Risk

- 본 문서에서 설계한 최종 구조는 Multi-Core를 통한 병렬화로 성능을 향상시킨다. 현재 Task 할당을 위한 Dispatcher Core는 하나만 존재하며, Dispatcher Core의 Task 할당 오버헤드가 클 경우, 시스템 성능이 저하될 수 있다. 이 경우, 더 큰 단위로 Task 샤딩을 통해, Dispatcher Core의 오버헤드를 줄이는 전략을 사용하여야 한다.
- 본 문서에서 설계한 최종 구조는 각 제조사마다 인터페이스가 통일된 스토리지 Driver를 제공하고 있음을 가정한다. 따라서 공통의 인터페이스를 제공하지 않는 신규 스토리지 Driver가 많아지면, Adapter 추가가 빈번히 발생할 수 있다. Adapter가 많아질 경우, Driver 수정을 통해, 이를 제거하는 것이 필요하다.