BLEEP : Universal tests for blockchains

Yonggon Kim

KAIST

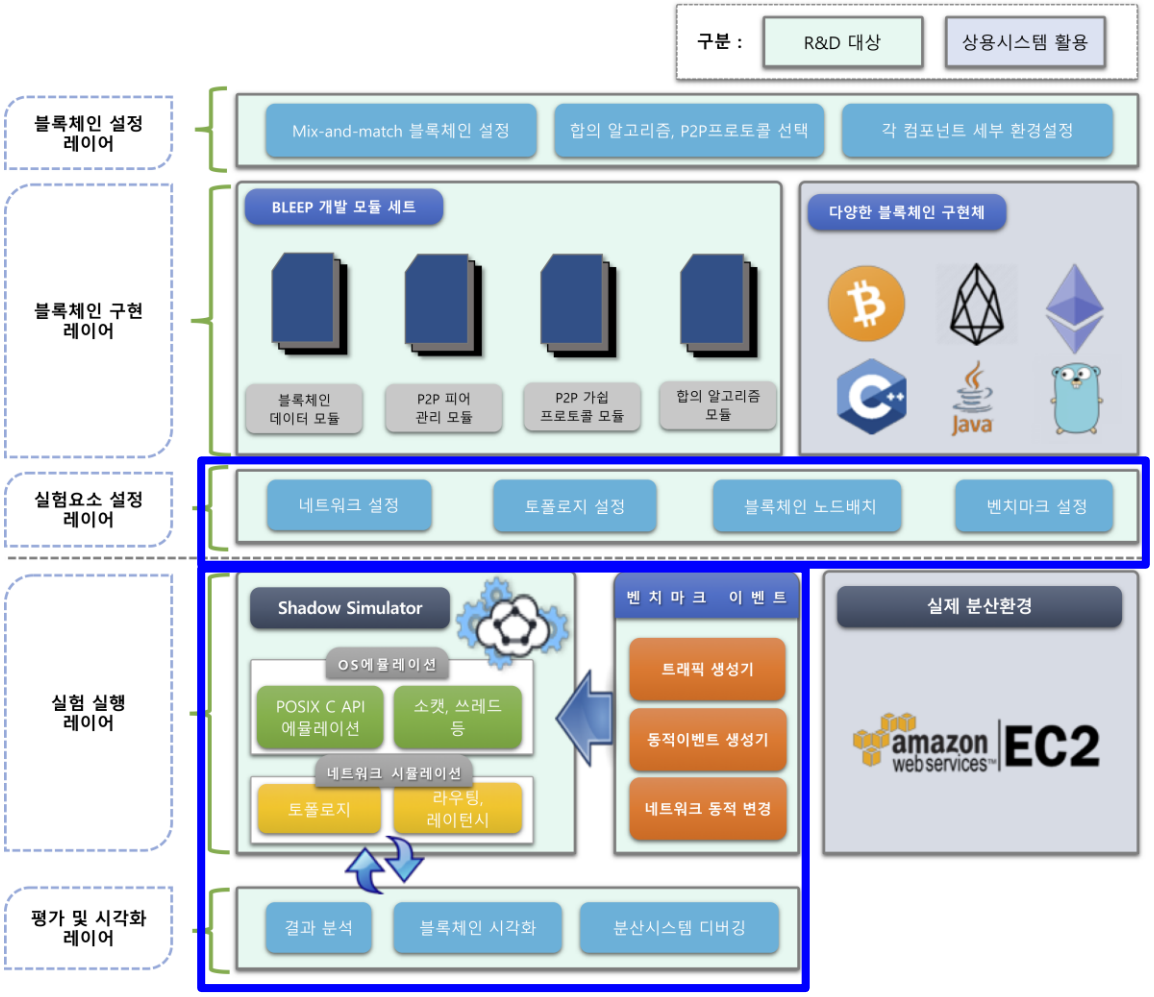
KAIST 한국과학기술원

BLEEP's goal in this course

- **목표 1. 블록체인 구현 템플릿 및 라이브러리 제공**
  - 모든 블록체인이 공통적으로 가지는 형태를 파악하여, 대다수 블록체인을 쉽게 구현해볼 수 있는 템플릿을 제공하는 것이 첫번째 목표
  - 또한 대다수 블록체인이 사용할 수 있는 공통 library 모듈을 제공하는 것이 목표
    - Data 모듈, 네트워크 통신, p2p 시스템, 등등

- **목표 2. 블록체인 테스트 플랫폼 제공**
  - 구현한 블록체인을 쉽게 테스트해보고 검증할 수 있는 방법을 제공하는 것이 두번째 목표
  - 여러가지 fault 를 생성해보고, 블록체인의 동작 결과를 검증
  - 벤치마크 형태로 테스트를 제공하며, 블록체인 구현체에 independent 한 자동화된 테스트가 가능하도록 테스트 프레임워크를 제공

BLEEP : Architecture



Today's topic

- 지난 수업에서는, 블록체인 구현 템플릿 및 라이브러리를 제공하기 위해 블록체인의 가장 추상화된 형태에서부터 시작
  - 추상화된 모델을 바탕으로 어떻게 실제 동작하는 구체적인 library component 까지 디자인하게 되었는지에 대해 설명 (top-down)
- 그렇다면, 어떻게 구현한 블록체인을 쉽게 동작시키고 테스트해볼 수 있을까?
  - 수많은 node 가 동작하는 distributed system 의 설정 및 동작
  - 수많은 node 가 동기화 되지 않은 네트워크 위에서 독립적으로 동작함, 따라서 분석 및 디버깅 조차 어려운 환경
  - 테스트는 어떻게 디자인 되어야 하며, 블록체인의 correctness 는 어떻게 formal 하게 검증할 수 있는가?
- 오늘은, **BLEEP**을 통해 어떻게 블록체인을 범용적으로 테스트할 수 있는 플랫폼을 제공할 수 있는지에 대해 얘기할 것이다
  - 가장 기반이 되는 기술부터 차례로 설명(bottom-up)

Outline

- 네트워크 시뮬레이터 **(Shadow)** 의 컨셉 및 시뮬레이션 아키텍쳐
- **Shadow** 가 어떻게 실제 **application** 들을 동작시키는지
  - App memory management
  - App execution management
- **Design & implementation of universal blockchain tests on Shadow**
  - 블록체인 추상 모델을 활용하여, shadow 상에서 블록체인 테스팅 기능을 어떻게 구현했는지에 대해 설명

Shadow

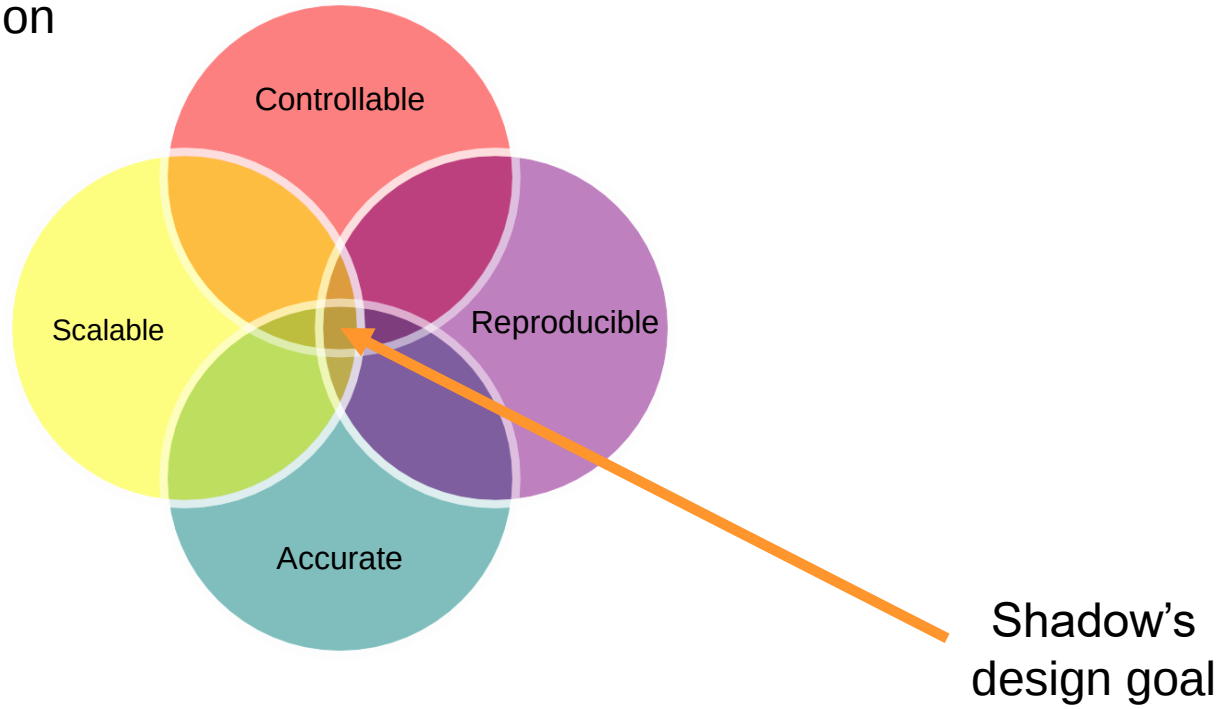
- **Open-source network simulator / emulator hybrid**
- 시뮬레이션으로 동작시키는 네트워크 토폴로지 위에서 실제 어플리케이션 **(binary executable)** 들을 직접 동작시킴
- **Efficient, scalable, deterministic, accurate, and more!**

Shadow : motivation

- 네트워크 어플리케이션에 대한 연구 및 개발의 전문화
- 소프트웨어의 기능 및 보안 문제에 대한 평가
  - 실제 사용자 및 실제 네트워크에 영향을 주지 않는 평가 가능
- 실제 네트워크에 **deploy** 를 하기 전에 종합적인 평가 진행 가능
- 새로운 어플리케이션에 대한 시뮬레이션 가능

Shadow : Experimentation option

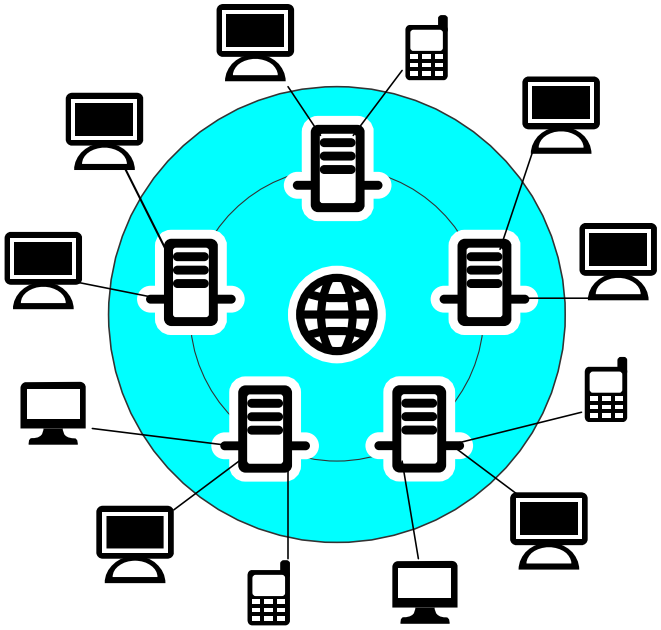
- 네트워크 어플리케이션 동작 평가를 위한 여러가지 옵션이 존재
  - Live network experiments
  - Testbed network experiments
  - Network simulation
  - Network emulation



Live Network Experiments

- **Experimenting in a deployed system**

- Pros
  - Most realistic
  - The target environment



- Cons
  - Hard to manage/debug
  - Lengthy deployment
  - Security risks

Testbed Network Experiments

- **Experimenting in a distributed testbed (e.g. PlanetLab)**

  - Pros
    - Close to target environment
    - Runs on the Internet or uses Internet protocols
  - Cons
    - Hard to manage and debug
    - Doesn't scale well in low-resource environs
    - Can be hard to model network properties

Network Simulation

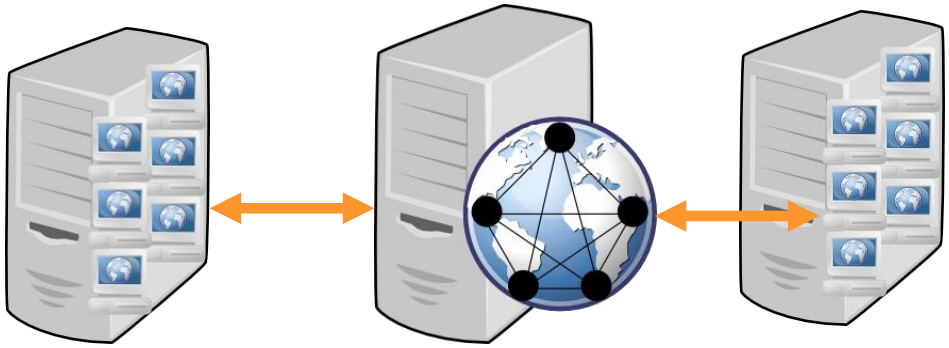
- **Experimenting with network simulators (e.g. NS3)**

- Pros
  - Deterministic (reproducible)
  - Can model various network properties
  - Can scale very well

- Cons
  - Application model can be too abstract
  - Abstractions can lead to inaccurate results

Network Emulation

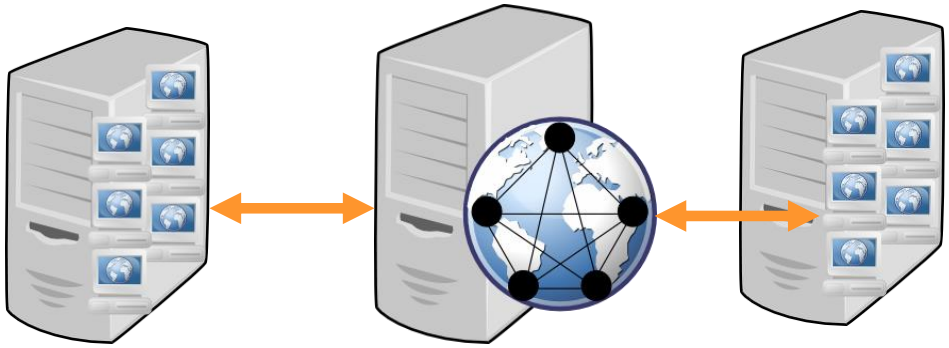
- **Between the extremes are network emulators, which run many instances of the relevant software on a few machines**
  - E.g., the ExperimenTor, NetMirage
- 이와 같은 툴들은 **delpoyed network** 와 비교했을 때, 훨씬 컨트롤이 쉬우며 **physical resource** 에 대한 제한도 상대적으로 적다



Network Emulation

- 하지만 여러 문제가 존재한다

- **Realtime** 으로 여러 어플리케이션을 스케쥴링해야 하므로, 어플리케이션이 동작하는 환경이 **deployed** 환경과 많이 달라질 수 있다
  - Ex) 에뮬레이터에 100 개의 어플리케이션을 돌린다면?
  - 결과적으로 시뮬레이션에 비해 scalability 가 매우 떨어짐
- **Emulation** 의 결과가 **non deterministic** 하기 때문에 실험 결과를 재현하는 등의 일이 불가능



Introducing Shadow

- **Shadow discrete-event network simulator** 는 앞선 여러가지 실험 옵션들에서 장점만을 조합하고자 디자인되었다
- **Realism v.s. Control**
  - 어떻게 실제 deployed network 과 같은 real 한 실험을 하면서도, 쉽게 (실험 비용, 실험결과 등을) control 가능한 실험 환경을 제시할 것인가?
- **Best-of-both-worlds approach**

Shadow : simulator / emulator hybrid

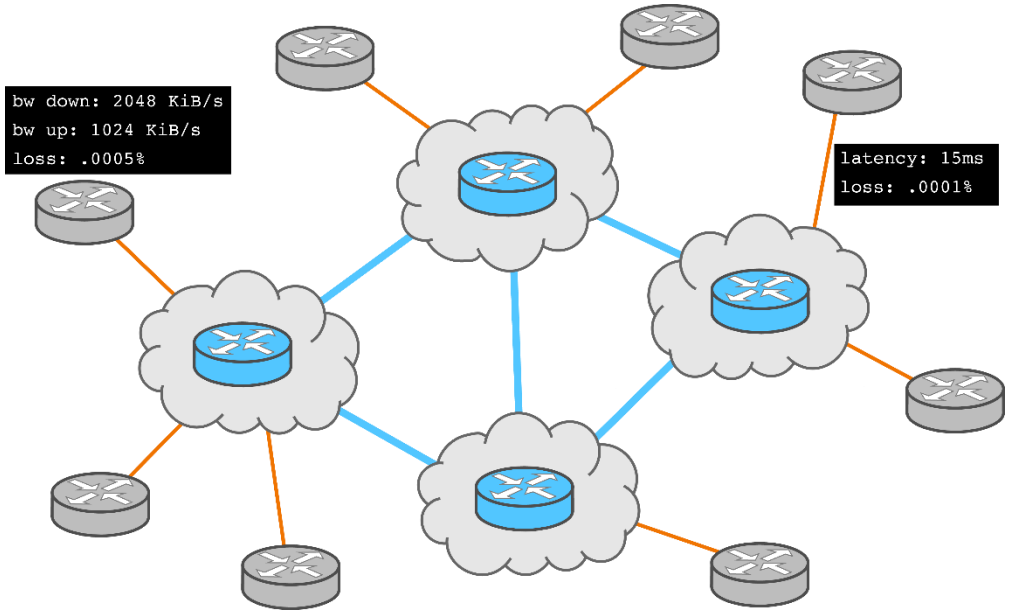
- **Shadow : Open-source network simulator / emulator hybrid**
  - Shadow 는 network simulator 이지만, application 부분을 추상화하는 것이 아니라, 실제 application binary 를 돌릴 수 있도록 구현되었다

- 모든 **network stack** 을 자체적으로 **simulation** 하는 **virtual clock** 에서 동작을 시킴

- 동시에, **Tor, bitcoin** 과 같은 실제 컴파일된 **application code** 를 시뮬레이션되는 네트워크 토폴로지 위에서 직접 실행시킴

Shadow : network simulation

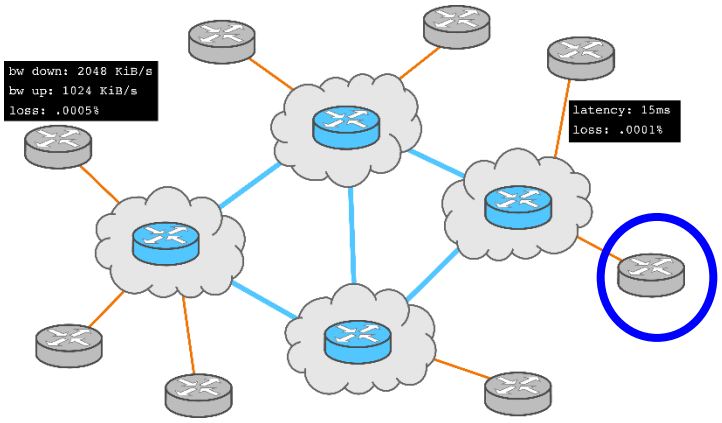
- **Routing, latency, bandwidth, TCP/UDP** 와 같은 네트워크 요소를 시뮬레이션
- **Load customized network model**
  - Recent shadow paper (CCS'2018) 은 실제 internet 과 매우 유사한 bandwidth, latency 환경을 재현해볼 수 있는 topology model 을 shadow 에서 돌려볼 수 있도록 관련 설정을 제공하고 있음



Shadow : OS emulation

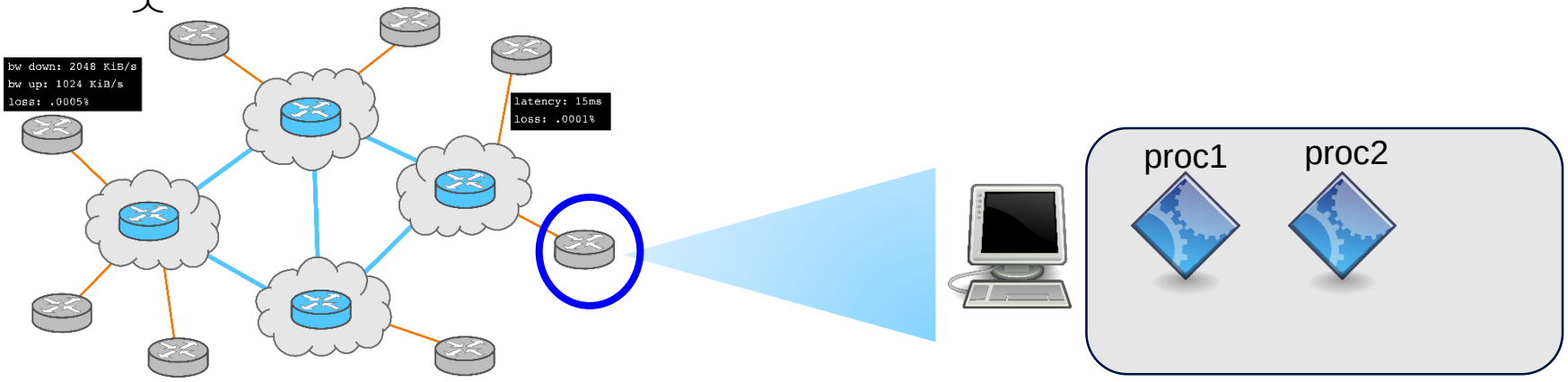
- **Shadow network** 위에서 동작하는 각 **host** 역시 시뮬레이션에 포함시켜 **control** 함
  - Virtual host, virtual processes, 심지어 virtual threads 까지.

- **Shadow** 가 **simulation** 하는 **network topology** 위에 어느 특정 **end point** 에서 어떤 일이 일어날 수 있는지 들여다보자



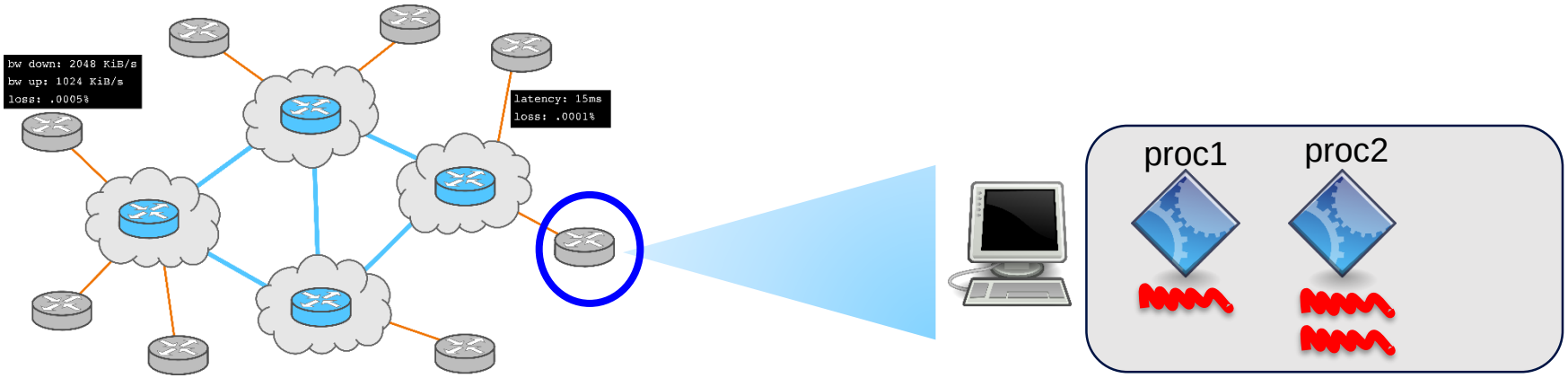
Shadow : OS emulation

- 예를 들어 한 **host** 가 해당 **network end-point** 에 연결되어 있다고 하자
- **Shadow** 는 이 **host** 가 구동하는 여러 **process** 를 **simulation** 할 수 있음
  - 즉, virtual process 지원 (shadow 상에서 동작하는 가상의 process)
    - 각 virtual process 들은 separate address space 를 가짐
  - 기본적인 process scheduling, queuing 구현 (OS의 기능들을 emulation)
  - 물론 이미 말했듯이, shadow의 virtual process 는 실제 binary 를 구동시킨 것



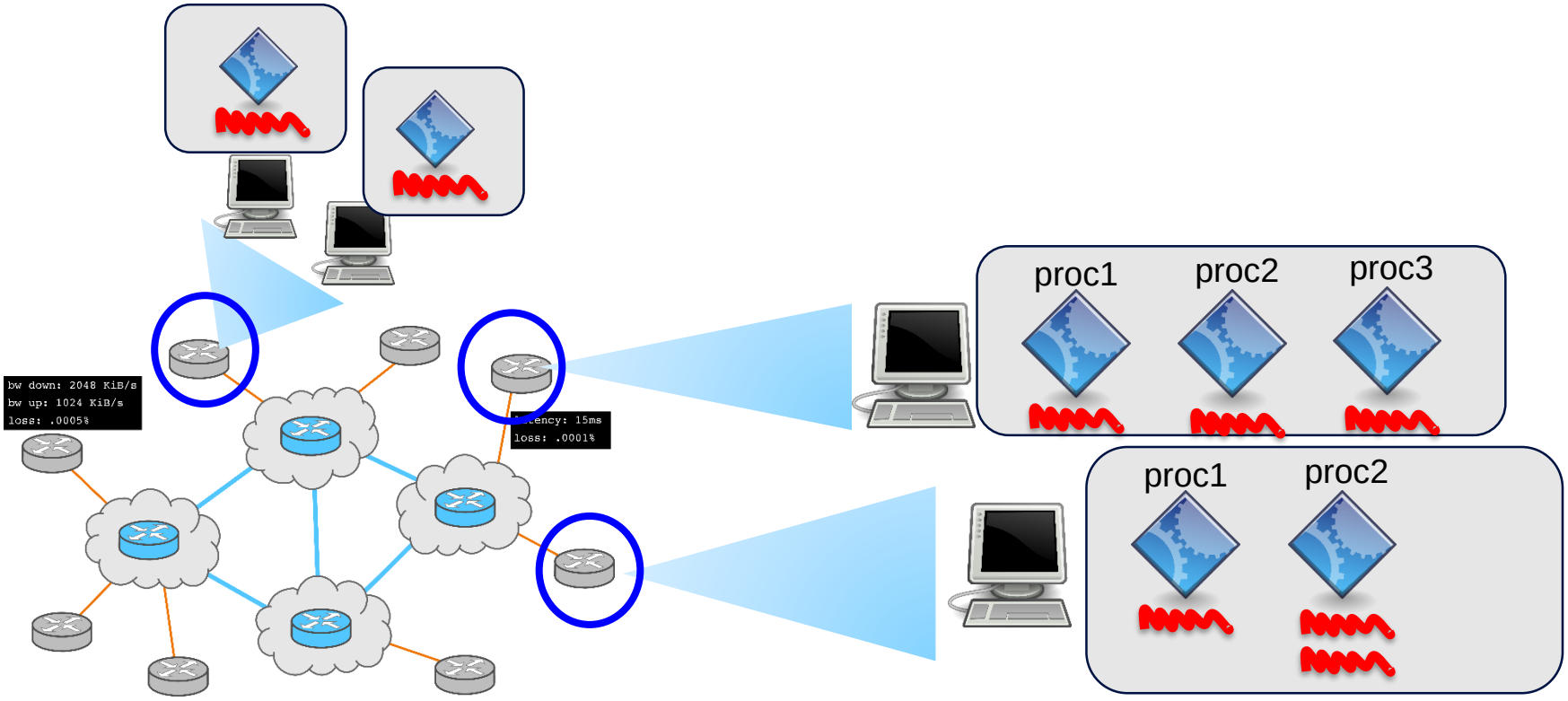
Shadow : pthread emulation

- **Shadow 는 각 process 가 동작시키는 thread 역시도 simulation 한다**
  - 즉, virtual thread 지원
  - Shadow 상에서 동작할 뿐이지, 실제 thread 의 동작과 매우 유사
    - Application 은 pthread system call 을 수정없이 그대로 사용 가능
    - 독립적인 program counter, stack 을 가지도록 구현
    - thread 간의 context switching emulation

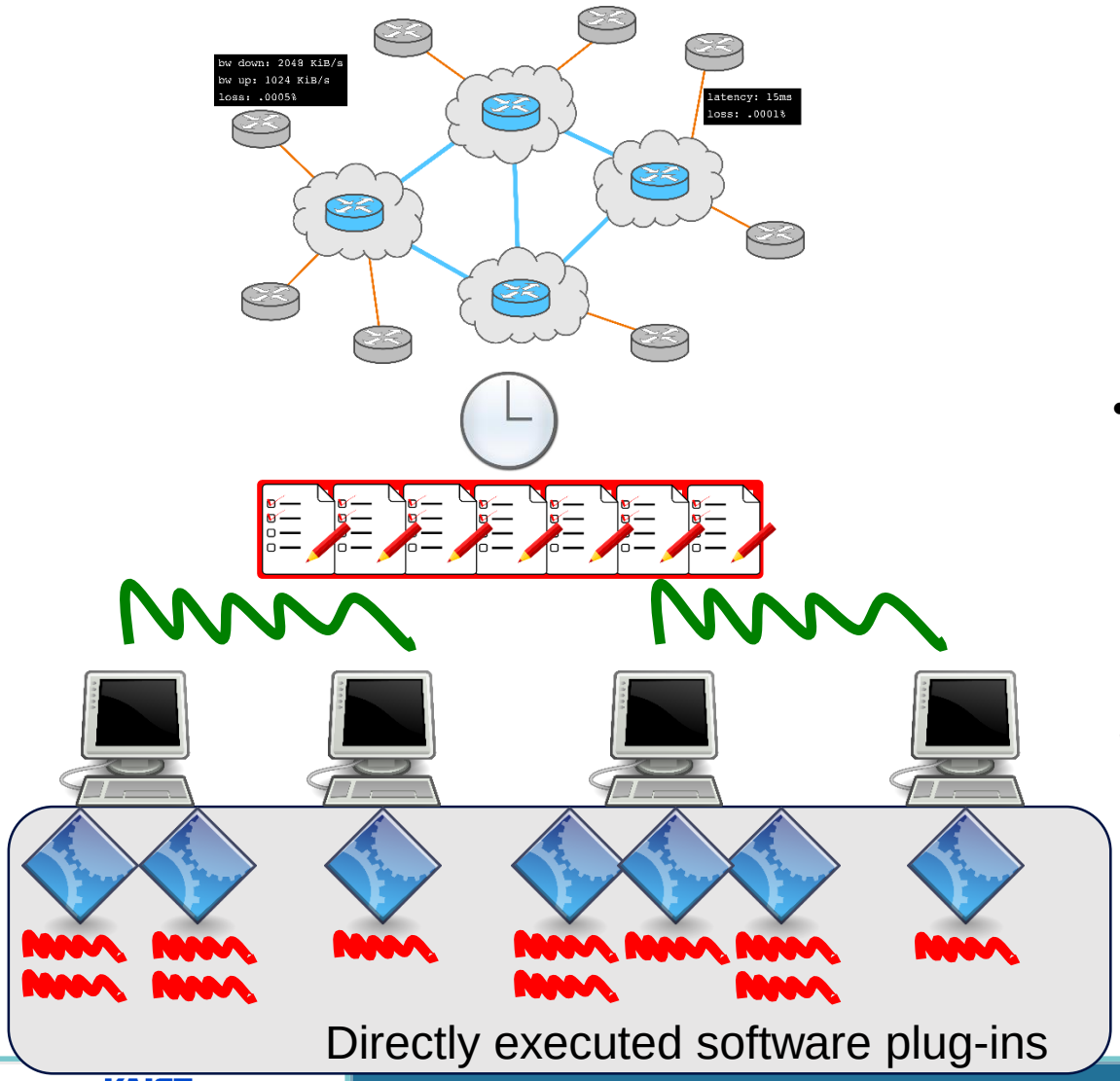


Shadow : Network simulation + OS emulation

- 이처럼 **host(OS), process, thread** 의 모든 동작들이 **shadow** 가 **manage** 하는 **network simulation** 위에서 이루어지고 **control** 된다



Shadow simulation architecture



- Network model
- Global simulation clock
- Discrete event queue
- Shadow worker threads
- Virtual hosts
- Virtual processes
- Virtual threads

Shadow 에서의 어플리케이션

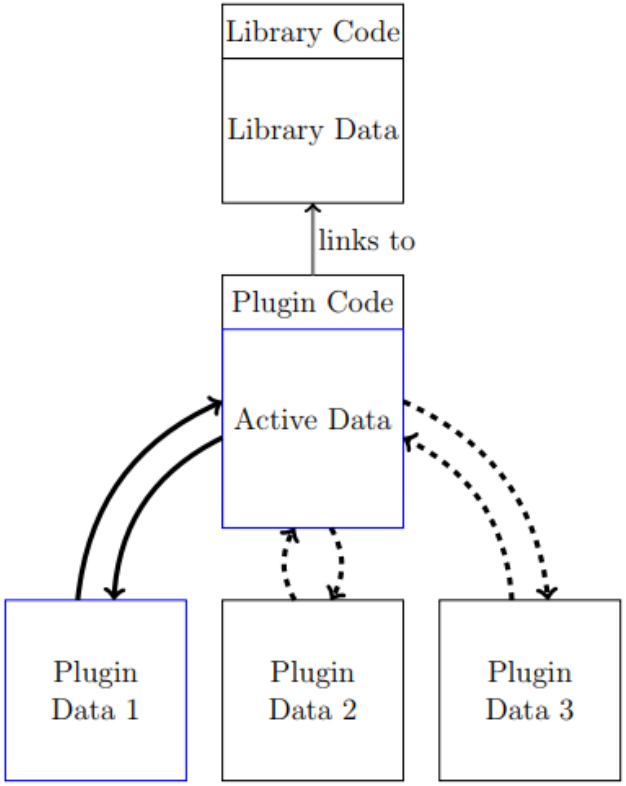
- **Directly executes software plug-ins**
- **What is plug-ins?**
- **How shadow controls its execution?**
- **Memory Management**
  - Independent namespace using customized loader
- **Execution Management**
  - Function Interposition

Background : Shadow plug-ins

- **Shadow** 에서 동작하는 **application binary** 의 형태
- **Application binary** 를 컴파일 할 때와 똑같은 개발 소스를 그대로 컴파일해서 만드는데, 단지 **shadow** 가 **load** 해서 실행시킬 수 있도록 **shared object** 형태로 **compile** 된 **binary** 임
- **Shadow** 는 그저 특정 타이밍에 맞춰 **plugin** 의 **main** 함수를 실행시키는 단순한 방법으로 **plugin** 을 실제 동작시키게 된다
  - 물론 여러 plugin instance 를 효율적으로 실행하기 위해, context switch 를 구현한다 (예 : plugin 이 i/o 에 관련된 함수를 호출할 때마다 실행하는 plugin 을 변경)

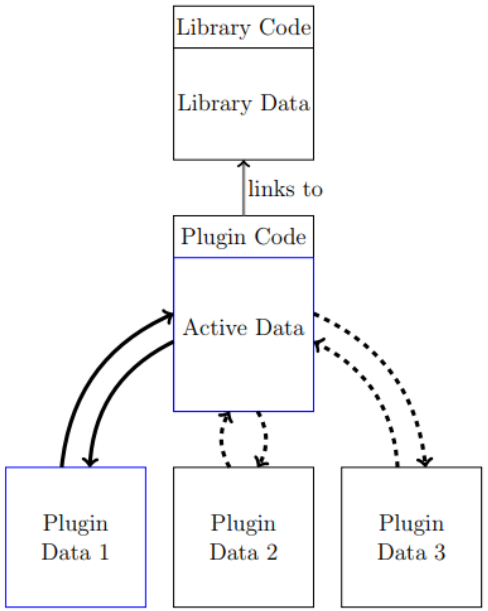
Shadow : App Memory Management

- 초창기 **shadow (2012년부터 v1.12.0 이전까지)** 에서는 **shadow** 는 **virtual process** 를 시뮬레이션하기 위해, **swapping state** 라는 개념을 사용하였다



Shadow : App Memory Management

- **Shadow** 는 여러 개의 **plugin instance** 를 동작시키게 됨
- 이때, 각각의 **plugin instance** 들은 변하지 않는 영역인 **code** 는 같이 공유하게 되고, 독자적으로 사용할 여지가 있는 **data** 영역을 각각 구분해서 사용하게 됨.
- **shadow** 에서 **plugin instance** 를 동작시키다가 다른 **plugin instance** 로 바꾸어 동작시키고 싶을 때, **plugin data** 영역만을 **update** 해주면 됨
- 이처럼, 각각의 **plugin instance** 가 가지는 **data** 영역을 **state** 라고 하며, 이를 바꿔가며 실행시키는 과정을 **swapping state** 라고 함



Shadow : App Memory Management

- 하지만 **swapping state** 는 큰 두가지 단점을 가지고 있었다
- 우선, **state** 를 **swapping** 하는 **operation,** 즉 **data** 영역을 **copy** 하는 과정이 **overhead** 가 크다
  - 만일 simulation 하는 plugin 의 data segment size 가 크다면, 전체 성능을 매우 떨어뜨릴 수 있음
- 그리고, **library** 에 대해서는 **state swapping** 을 적용하기가 쉽지 않음.
  - state swapping 을 하기 위해서는, library 를 그에 맞는 형태로 일일히 compile 시켜야함 (LLVM 을 이용한 manual 한 과정이 필요)
- 따라서 만일 **plugin instance** 들이 **library data** 를 **update** 하는 경우 문제가 생길 수 있다
  - ex) SSL library 에 어떤 cipher 를 사용하고 있는지에 대한 state variable

Shadow : App Memory Management

- **Usenix CSET'18**
  - 각각의 plugin instance 들은 code, data, linked library 를 모두 포함하는 독립적인 address space (namespace) 를 가지게 되었음
  - Drow-loader 라는 직접개발한 loader 를 이용해 plug-in 을 memory 에 load 할때, 각각의 instance 들을 독립적인 memory space 에 할당

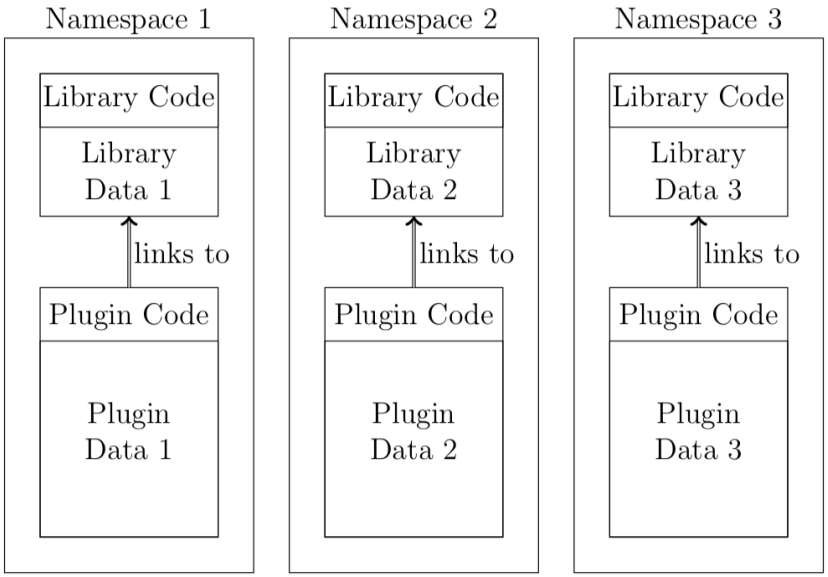


Figure 4.1: Our new design. Each instance of the plugin has its own dedicated namespace, which includes its code, data, and linked libraries.

Shadow : App Memory Management

- 메모리 **overhead** 이슈가 존재하지 않는가?

- **Drow-loader 는 shared memory mapping 을 이용함**
  - Mmap 시스템 콜을 이용해서, plugin 을 memory 에 load 함
  - Linux 에서 제공하는 COW (Copy-On-Write) 가 동작하게 됨

- **Linux COW**
  - 동일한 memory page 에 대해 하나의 physical memory page 를 유지
  - Update 가 발생할 때, update 가 발생한 page 에 대해서만 복사하여 새로운 복사본을 생성하는 테크닉
  - Application 의 동작에 상관없이 kernel 이 transparent 하게 지원하는 기능

Shadow : App Memory Management

- 이를 통해, 각각 독립적인 **address space** 를 가지되, 같은 내용의 메모리 **page** 에 대해서는 하나의 **physical memory** 만을 유지하는, 효율적인 **application memory** 관리가 가능하다

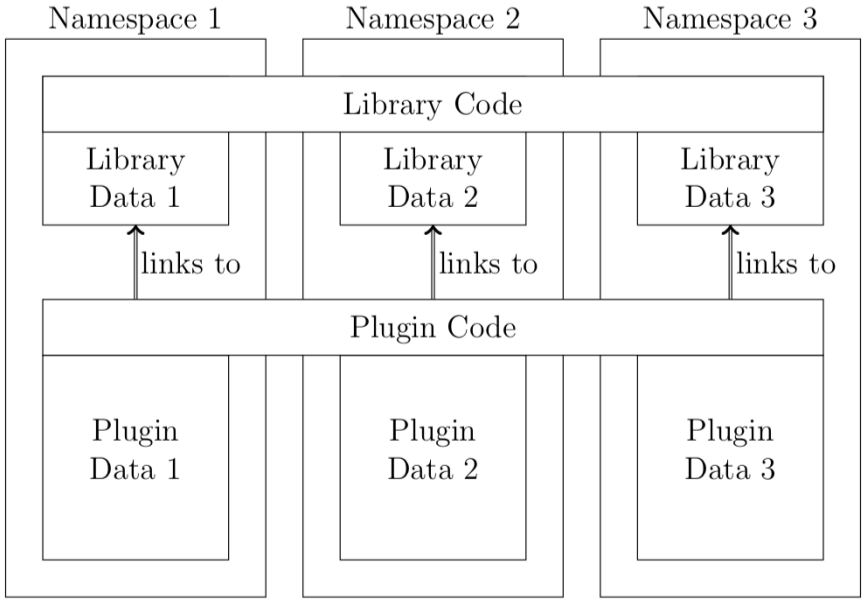


Figure 4.2: The memory layout found in Figure 4.1, from the perspective of the kernel. Each read-only segment, and any pages that have not been modified, that are mapped from the same file are backed by the same physical memory (in this case exemplified by the code of each shared object).

Shadow : App Execution Management

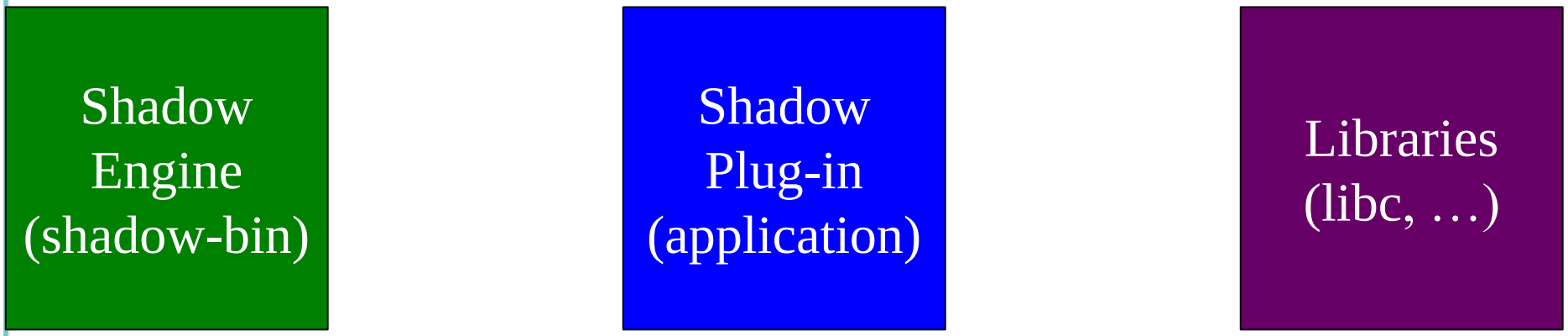
- **Shadow** 에서 동작하는 **plug-in** 은 **shadow** 가 시뮬레이션하는 환경에서 동작하는 **virtual** 한 **process**

- 네트워크 **i/o, file i/o, pthread** 등 다양한 기능들과 이를 위한 **system call** 을 어떻게 올바르게 동작시킬 수 있는가**?**
  - Ex) plugin이 socket open 을 요청할 때, 외부 네트워크와 연결하기 위한 socket 이 아닌 simulation 상의 socket 을 만들어야 함
  - Ex) Pthread_create 을 했을 때, 진짜 OS 가 preemptive 하게 scheduling 하는 thread 를 만들면 안됨

- **Function interposition**
  - Plug-in 이 library 함수를 호출할 때, shadow 로 hooking 해서, 알맞게 emulation 하고 simulation 할 수 있도록 함
  - 중요한 모든 libc 함수를 hooking 해서 shadow 내에 알맞게 구현함

Shadow : App Execution Management

- **Program layout**



Shadow : App Execution Management

- **LD_PRELOAD=/home/shadow/libpreload.so**

libpreload (*socket, write, …*)

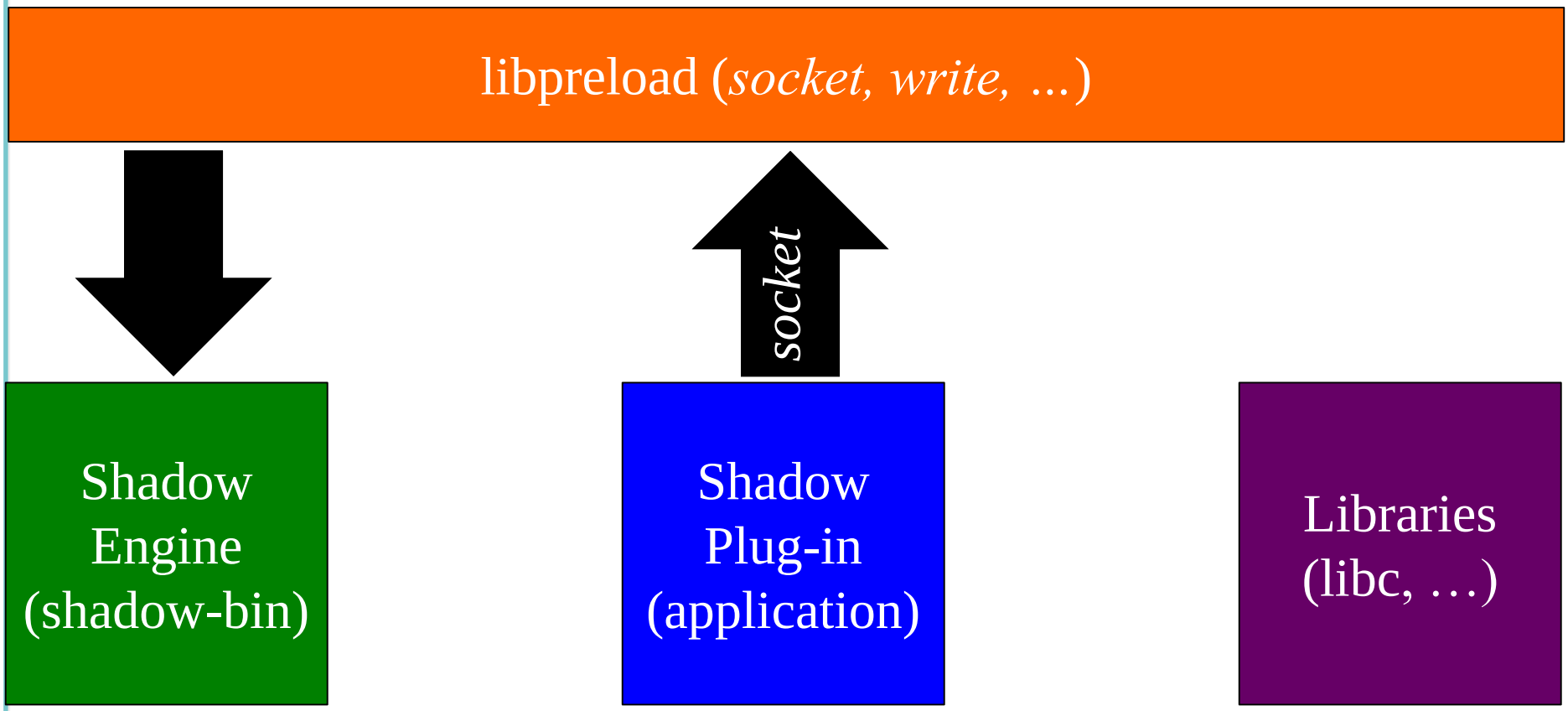
Shadow Engine (shadow-bin)

Shadow Plug-in (application)

Libraries (libc, …)

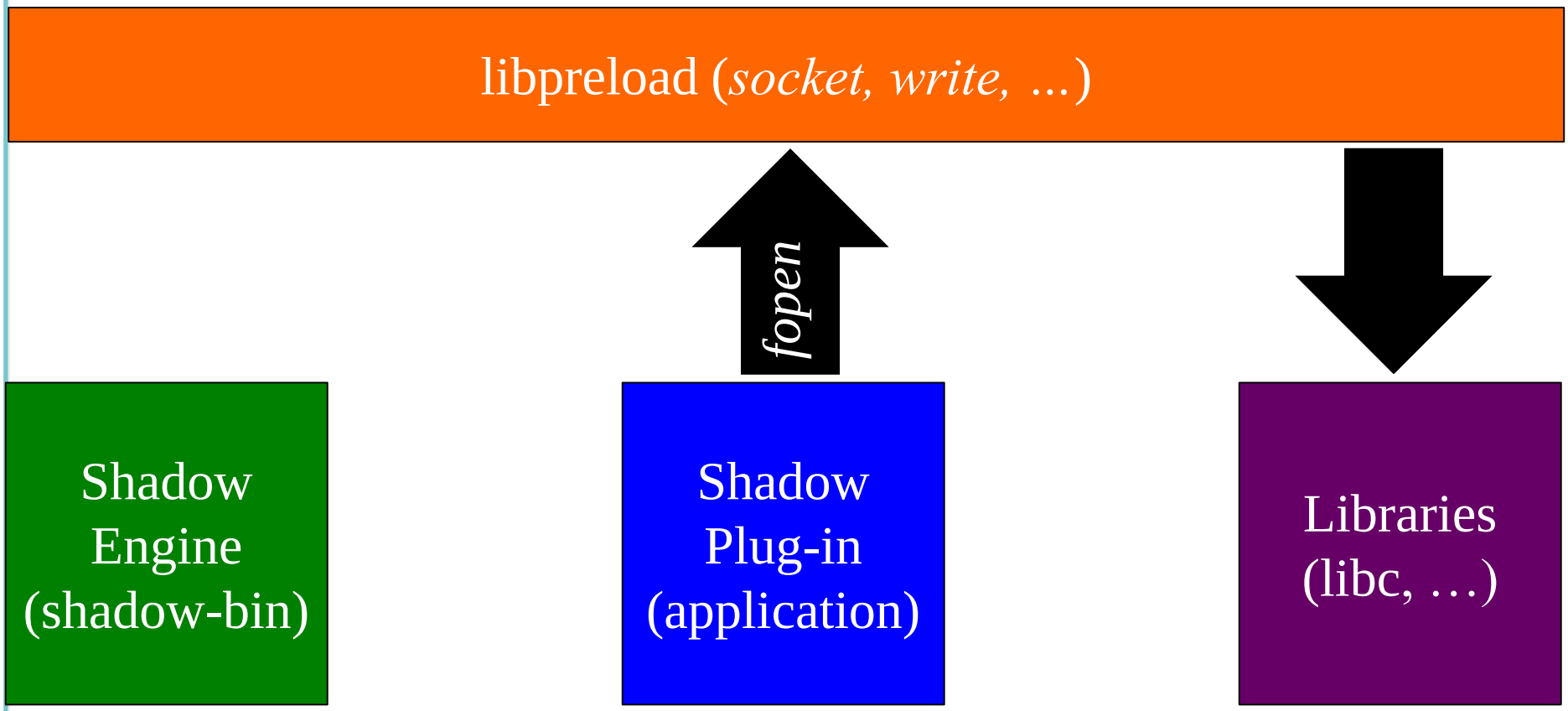
Shadow : App Execution Management

- **LD_PRELOAD=/home/shadow/libpreload.so**



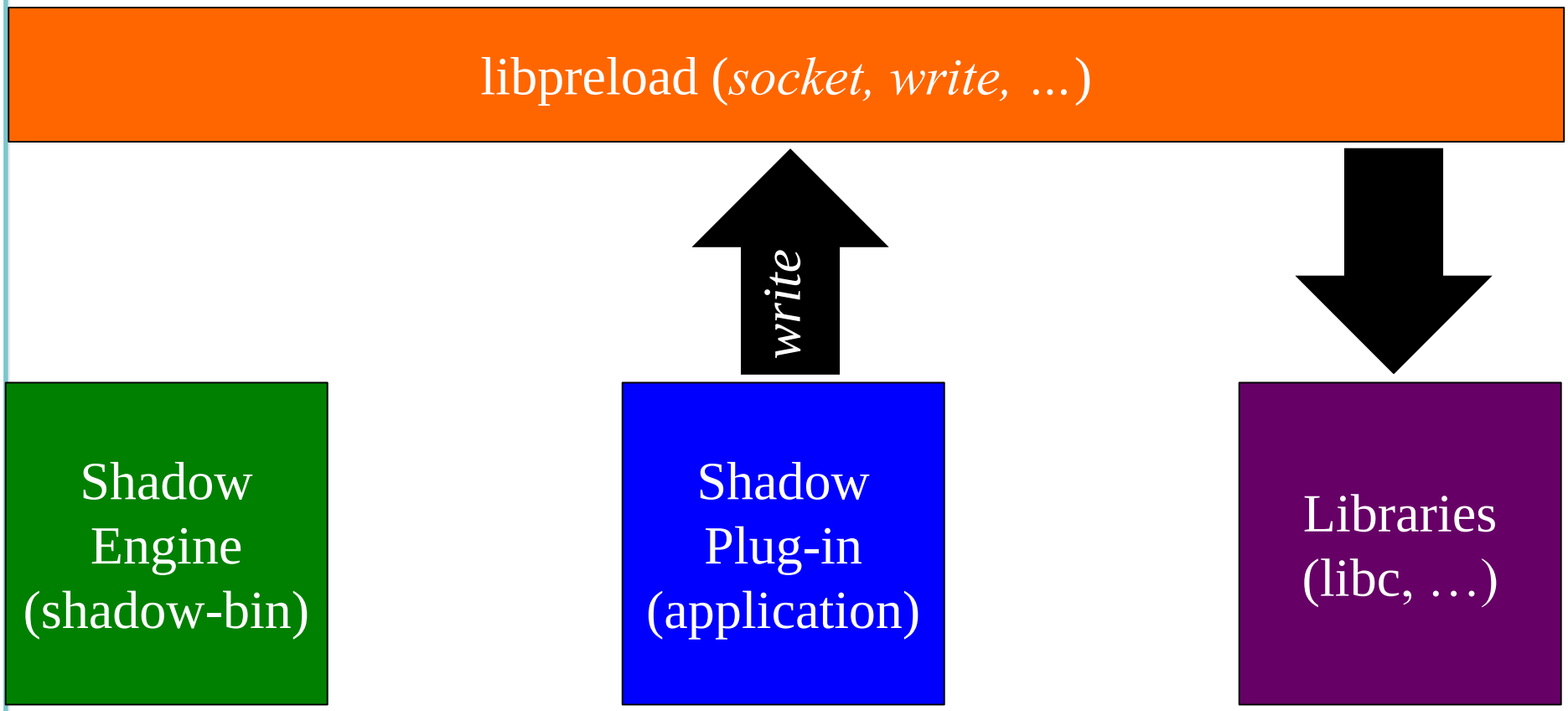
Shadow : App Execution Management

- **LD_PRELOAD=/home/shadow/libpreload.so**



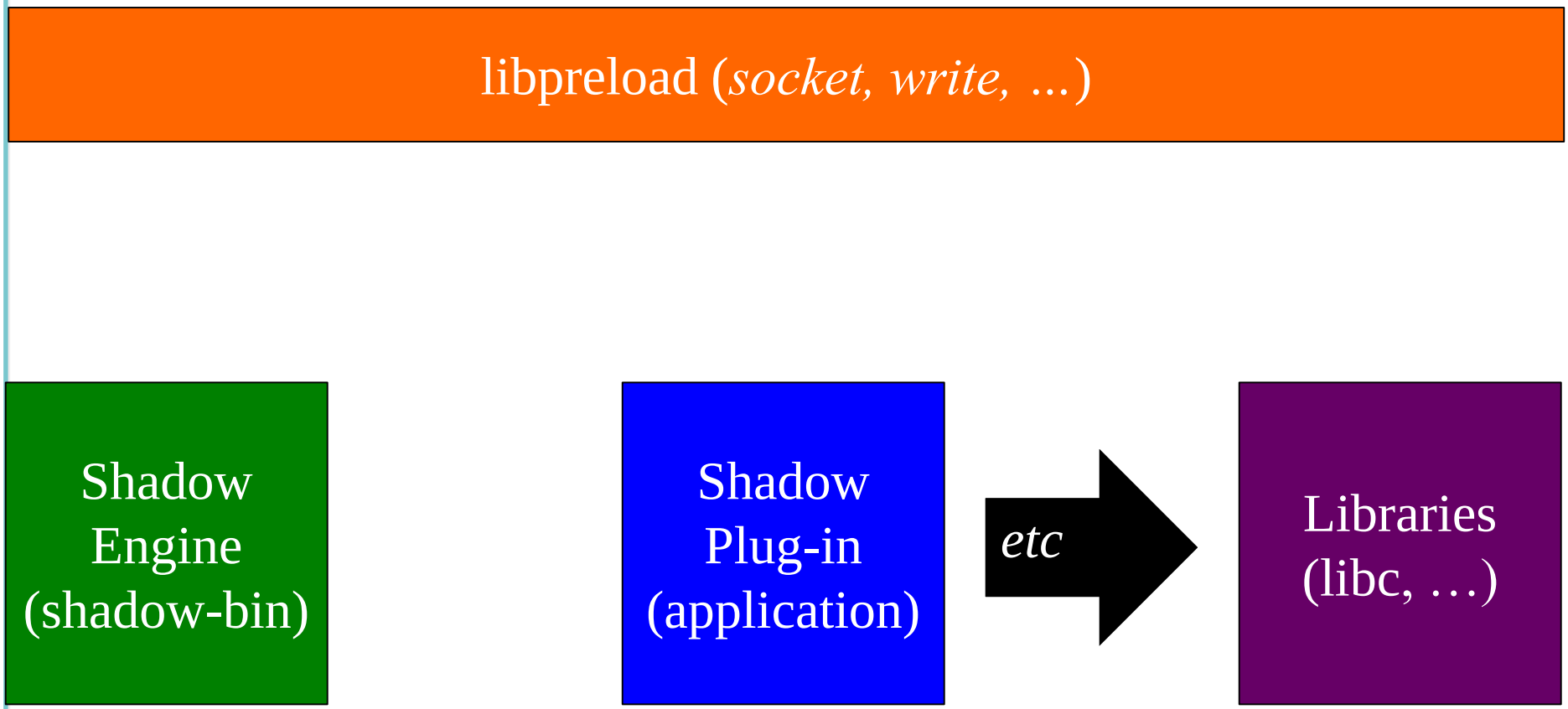
Shadow : App Execution Management

- **LD_PRELOAD=/home/shadow/libpreload.so**

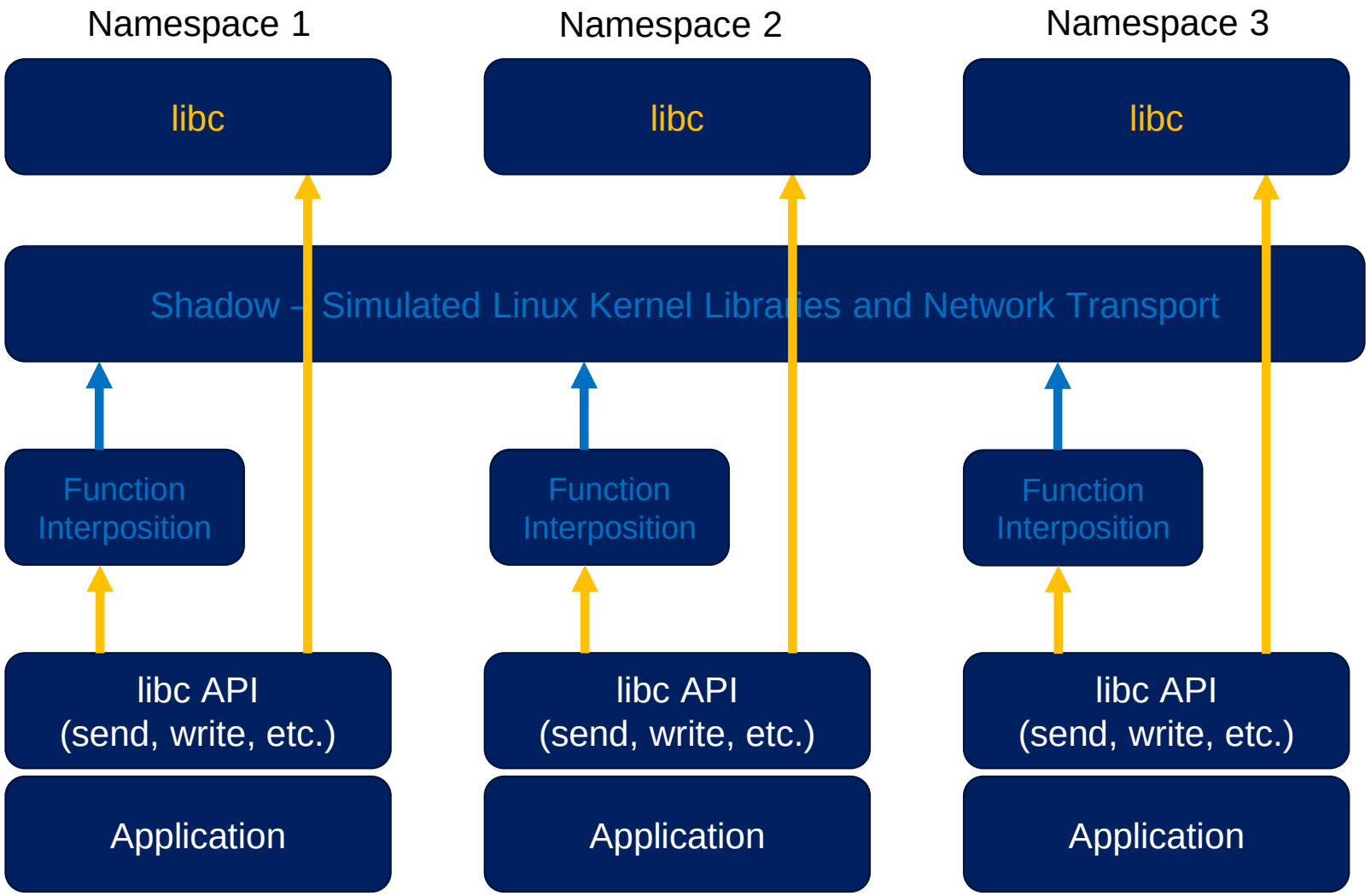


Shadow : App Execution Management

- **LD_PRELOAD=/home/shadow/libpreload.so**



Shadow : App Execution Management

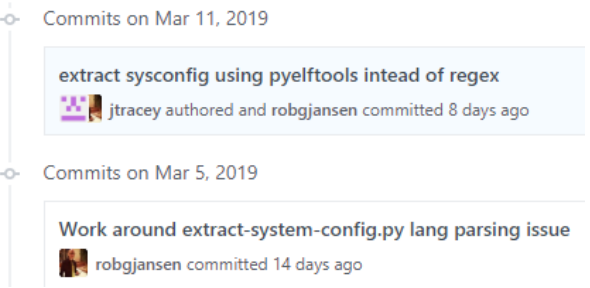


Shadow : Conclusion

- **The important design goals for large-scale distributed systems experiments**
  - Experimental control : Determinism yields repeatable / reproducible experiments
  - Realistic : execute software (not an abstraction), 그리고 libc, OS 와 같은 realistic system software stack 역시 반영
  - Scalable : can run studied system at scale
- **Shadow is network simulator with above design goals**

- **Efficient application management on shadow**
  - Custom loader & independent namespaces
  - Proper implementation of libc (system call) using function interposition

- 실제로 **serious** 하게 연구가 계속 되고 있음
  - Usenix'19, IMC'18, CCS'18, NDSS'18, …



Blockchain testing on Shadow

- **Shadow architecture 및 shadow 상에서의 app (plug-in) management 에 대해 설명을 하였음**
- **Leveraging shadow for testing blockchain**
  - BLEEP 은 shadow 를 fork 하여 BLEEP 의 simulation platform 으로 활용하고 있음
  - Shadow 덕분에 전체 블록체인 네트워크를 single host 에서 시뮬레이션 가능
    - Scalability
  - 네트워크에서 동작하는 모든 host, process, thread 가 shadow 라는 하나의 프로세스 상에서 컨트롤 됨
    - Controllable experiments

- **단지 shadow를 이해하고 사용한다는 것만으로도, blockchain 을 test 하기 위한 최소한의 기반은 갖추어진 셈**
  - 하지만, 어떻게 test 를 할 것인가?
  - 각 node 에서의 log 출력 + 분석을 반복하는 것이 최선인가?
  - Blockchain 추상화를 통해 구현을 위한 정형화된 component 를 뽑아냈던 것처럼, blockchain test 를 위한 공통 분모는 없을까?

Blockchain testing on Shadow

- **BLEEP 의 핵심 goal**
  - “Universal tests for blockchains”
  - 모든 블록체인에 공통적으로 적용 가능한, 정형화된 테스팅 방법 제안
  - 대다수 블록체인에 적용가능하며, 제각각의 테스트를 위해 따로 디자인하고 구현할 필요없이 사용할 수 있는 테스트 벤치마크 설계 및 제안

- **예를 들어 universal tests 가 될만한 것은 어떠한 것들이 있을까?**
  - 노드에 Input 을 임의로 발생시켜서 동작시키기 (input traffic test)
  - 다양한 네트워크 topology 테스트
  - 노드 간의 socket 연결 과정 디버깅
  - 주고 받는 다양한 message (consensus, gossip)에 대한 디버깅/분석 및 시각화
  - 생성하는 output (blockchain) 에 대한 correctness 검증
    - Agreement, Validity, Termination

Blockchain testing on Shadow

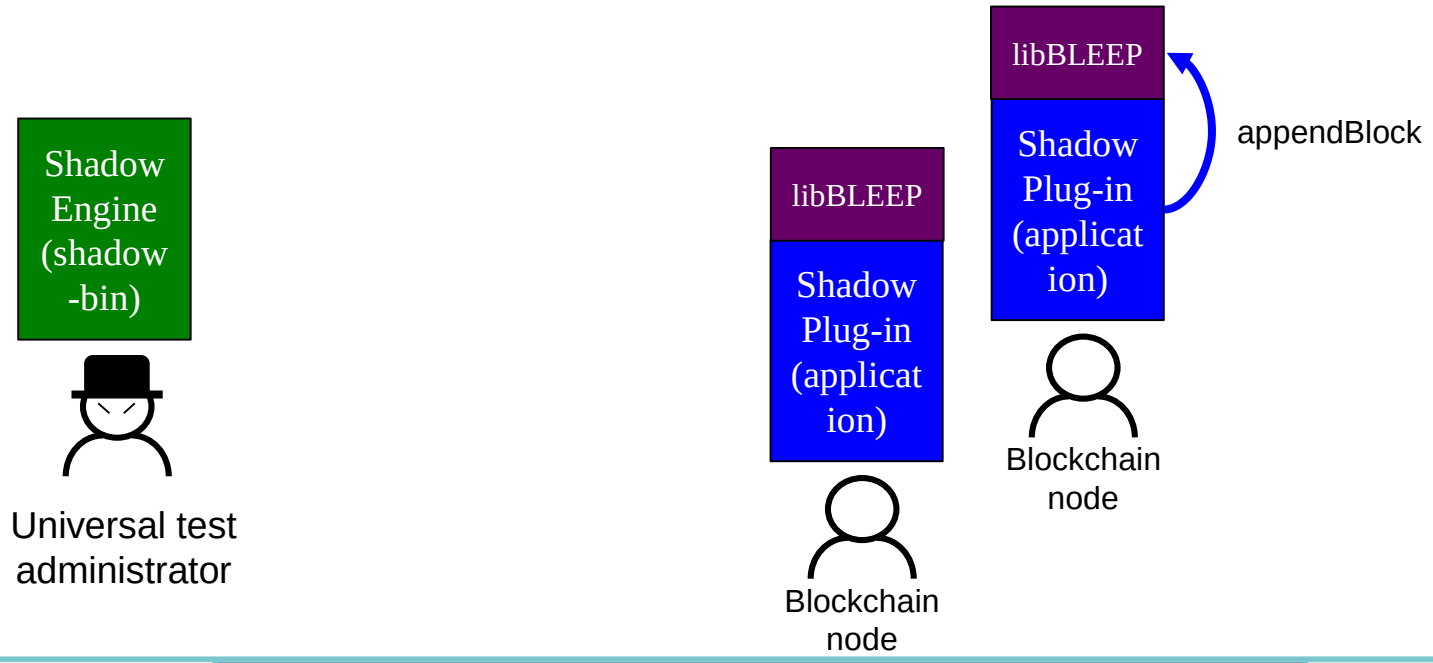
- 우리는 이미 대다수 블록체인의 동작을 정의하기 위한 **formalization** 을 정의하였다
  - Input (transaction), message passing, output(blockchain), correctness 등
- **Key idea :** 똑같은 **formalization** 및 **definition** 을 이용해서 블록체인에 대한 **universal test** 를 정의할 수 있다
- **Define BLEEP's universal tests for blockchains**
  - Input control (transaction injection) -> 다각화된 Traffic generation & test
    - Blockchain system 이 다양한 input 에 대한 어떻게 반응하는가?
  - Message passing verification -> Distributed Protocol에 대한 debugging
    - Blockchain system 의 message passing 과정이 올바르게 동작했는가?
  - Output correctness check -> Blockchain 정합성에 대한 자동화된 verification
    - Blockchain system 이 올바른 output 을 생성했는가?

Blockchain testing on Shadow

- **Universal tests** 를 어떻게 구현할 것인가**?**
- 우리는 이미 동일한 **formalization** 을 기반으로 핵심 컴포넌트에 대한 모듈들을 구현하였다
  - Input structure (transaction), input control (shadowPipe)
  - Message passing (socket, message, peerlist)
  - Output structure (blockchain)
- **BLEEP library** 의 **component** 에 테스트를 위한 기능만 구현한다면, **BLEEP library** 를 이용해 구현한 모든 블록체인은 같은 테스트 기능 **(**즉, **universal tests)** 를 별도의 오버헤드 없이 사용할 수 있다

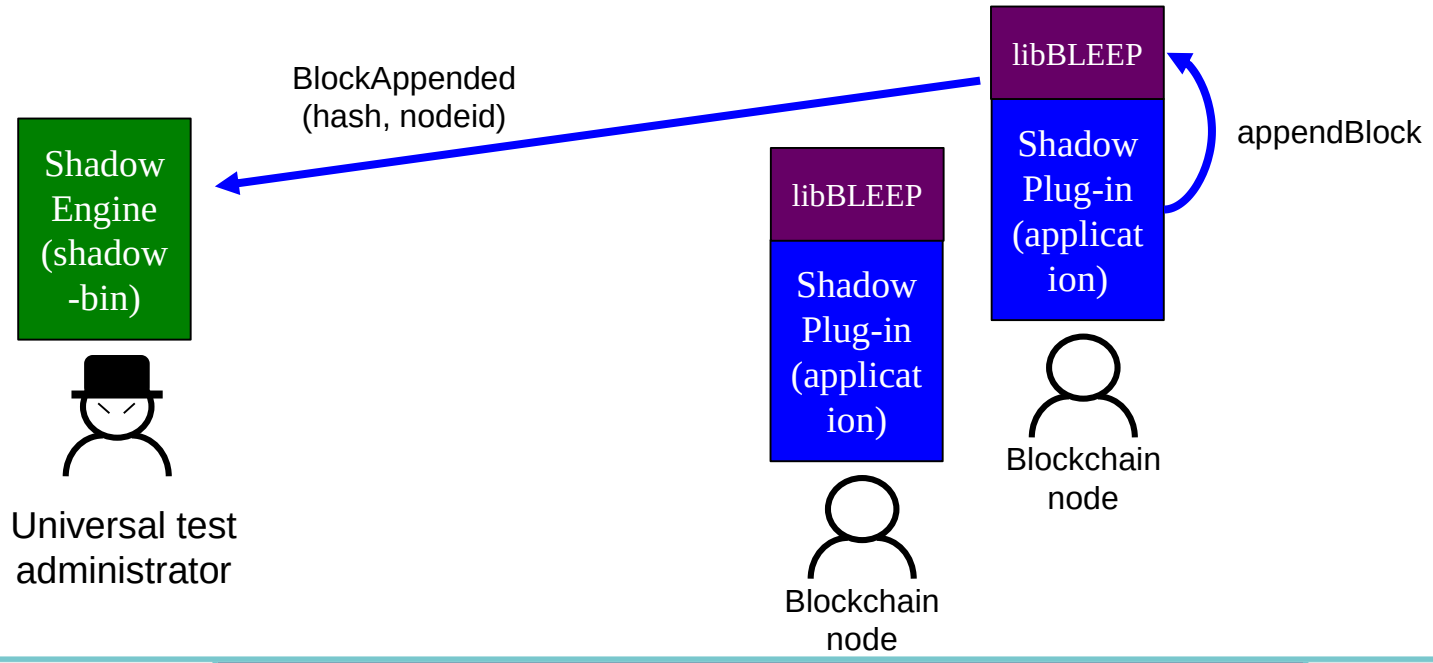
Implementing universal tests on BLEEP library

- **BLEEP library** 에 구현될 **universal test** 의 간단한 예제를 들어보자
  - Test 목적 : 생성하는 blockchain 에 대한 자동화된 agreement verification
- **Blockchain application** 은 합의의 결과로 **valid** 한 **block** 을 만들었을 때, **BLEEP library** 의 **ledgerManager** 컴포넌트를 이용해 **blockchain output** 에 **block** 추가를 요청할 것이다



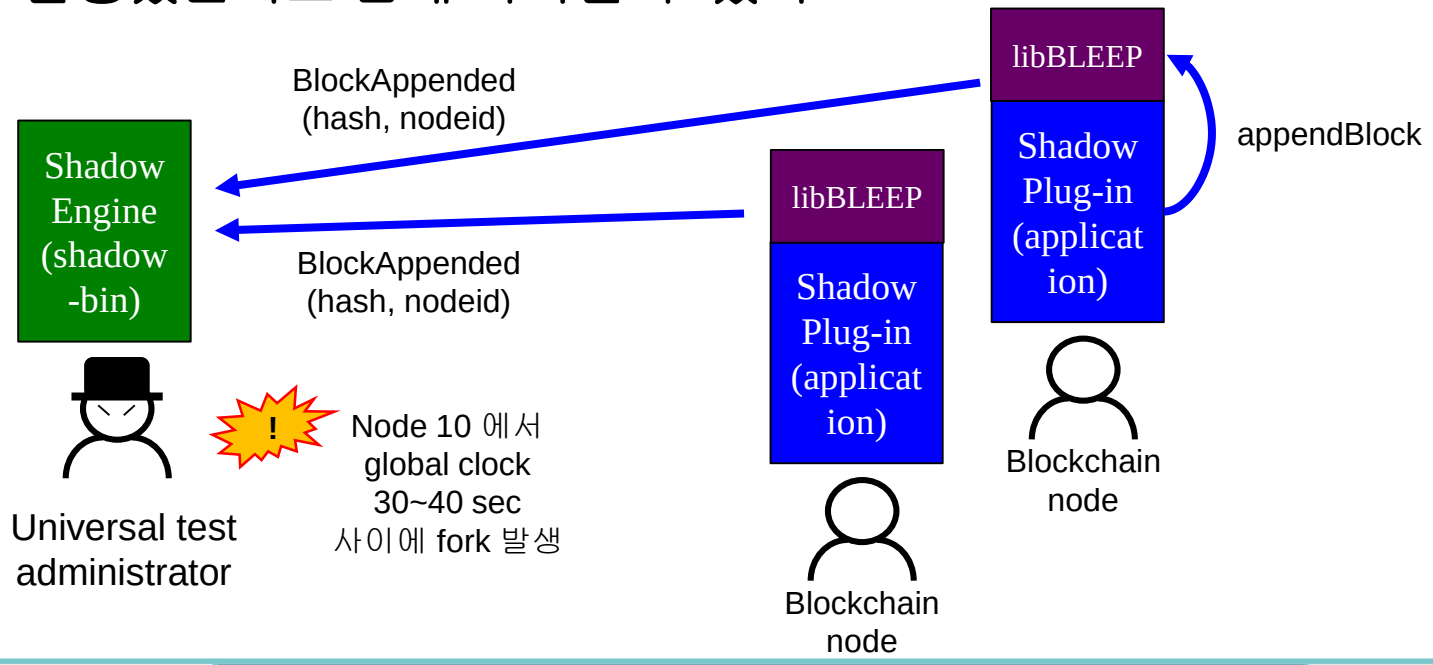
Implementing universal tests on BLEEP library

- **ledgerManager** 는 **appendBlock** 을 통해 들어온 **block** 을 자체적으로 관리하는 **blockchain data structure** 에 추가하는 역할을 우선 기본적으로 수행할 것이다
- 그리고 **universal test** 를 위해, 새롭게 **blockchain** 에 추가된 **block** 의 정보를 **shadow engine** 에 전달한다



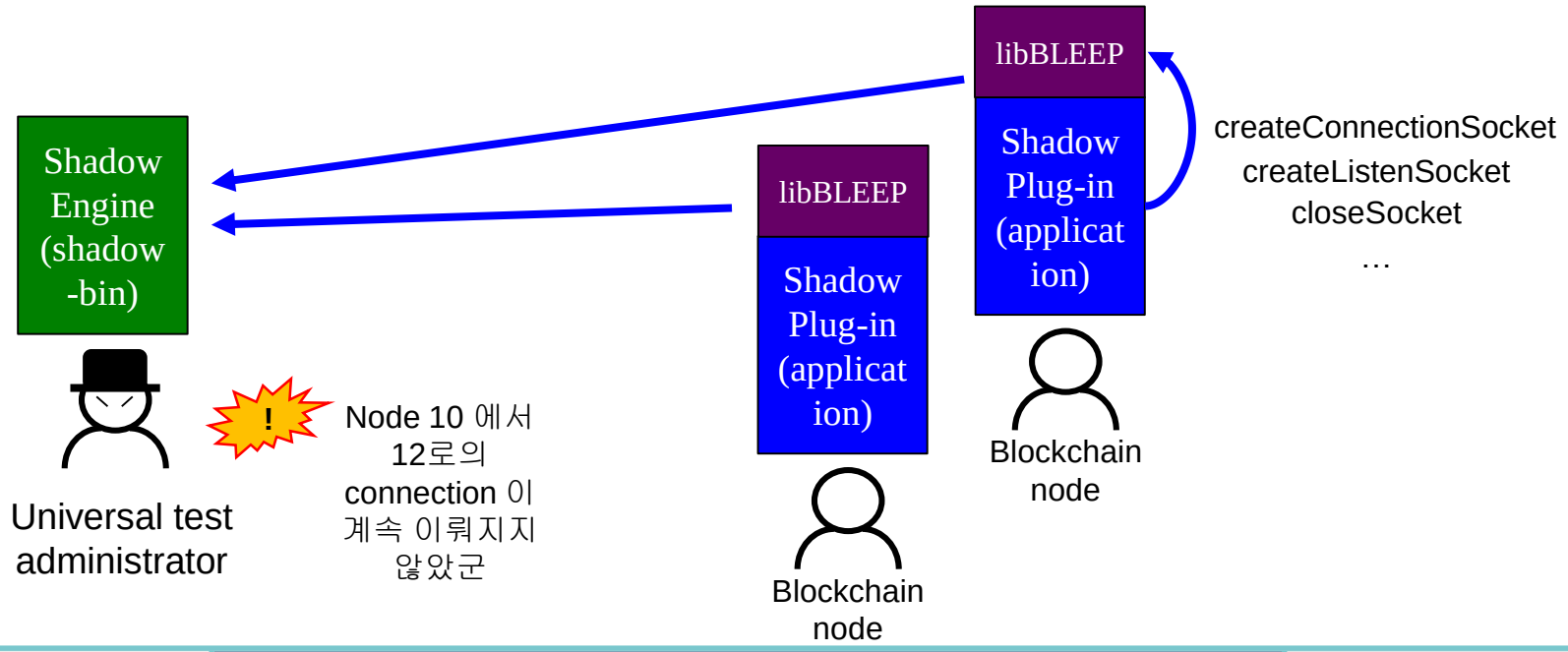
Implementing universal tests on BLEEP library

- **Shadow engine** 은 이렇게 모든 **node** 에서 **block** 이 생성될 때마다 관련 정보를 수집할 수 있고, **block** 이 생성되는 정확한 **global clock, hash, node id** 등의 정보를 축적할 수 있다
- 이를 이용해 최종적으로 **fork** 가 발생하지 않고 **consensus** 가 잘 이루어졌는지 **(agreement)** 를 체크하는 것은 물론이고, 어느 특정 순간에 **fork** 가 발생했는지도 쉽게 파악할 수 있다



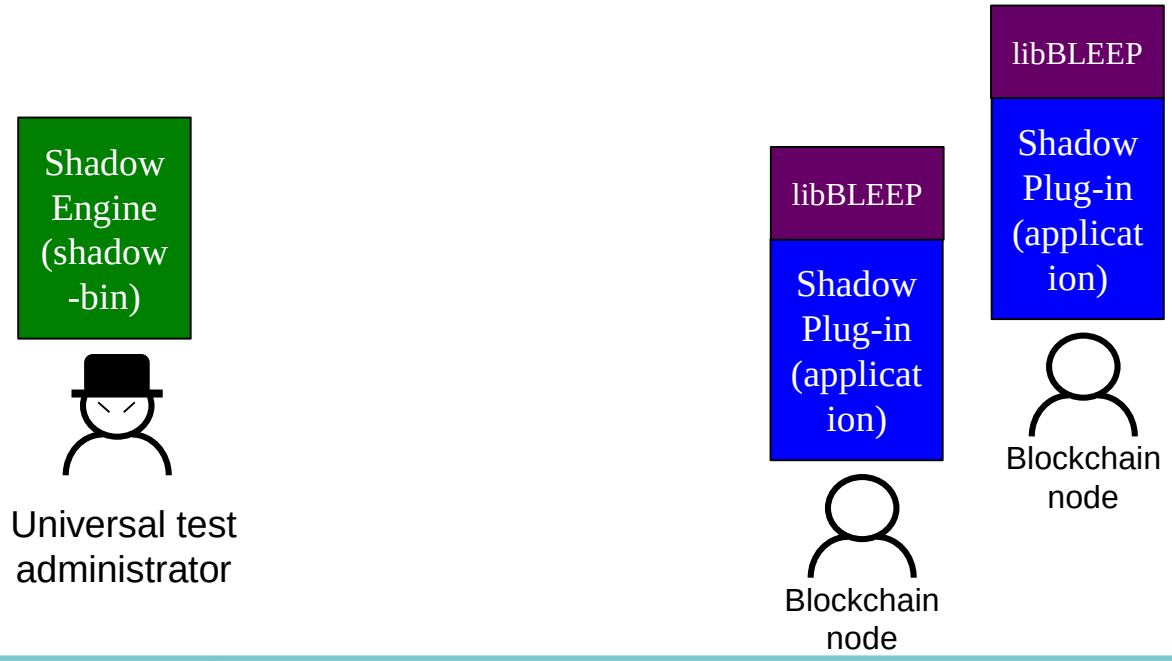
Implementing universal tests on BLEEP library

- **Node** 간의 **message passing** 을 위해 **socketManager** 를 이용해 **socket** 통신을 구현했을 경우, 어떻게 **Socket** 이 생성되고 연결이 이루어지고 끊어지고 있는지도 **centralized debugging** 이 가능하다
  - 일일히 socket 연결에 대한 log 를 찍어가면서 디버깅할 필요 없이, library 화된 모니터링 및 검증 과정을 이용해 자동으로 문제를 파악할 수 있다



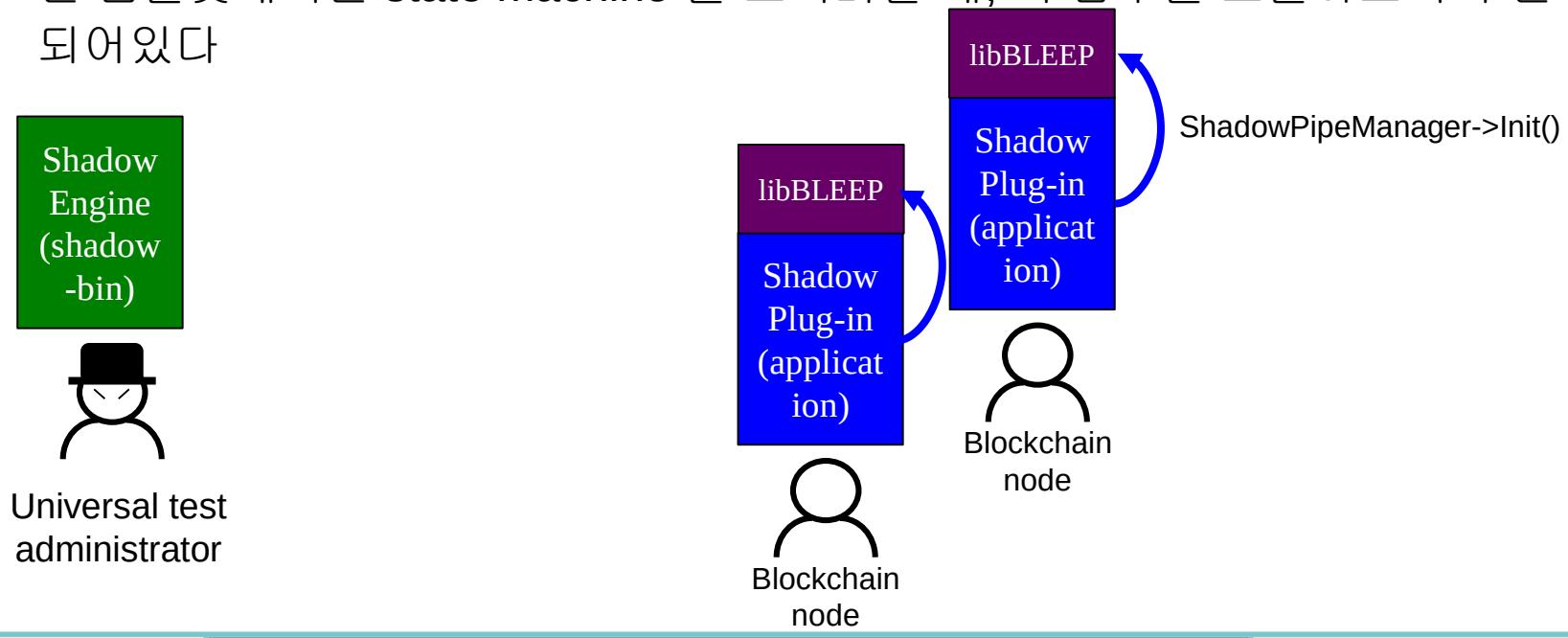
Implementing universal tests on BLEEP library

- 앞서 두가지의 예제는, 모두 **shadow engine** 이 **passive** 하게 **node** 의 동작들을 모니터링 하는 **test** 의 경우이다
- 반대로 **shadow engine** 이 각각의 **node** 를 **active** 하게 **control** 하는 경우도 존재한다
  - Ex) client 가 input 을 node 에게 주는 상황을 emulation, 특정 node 가 네트워크에 조인하여 다른 node 들에게 연결되는 시점을 control



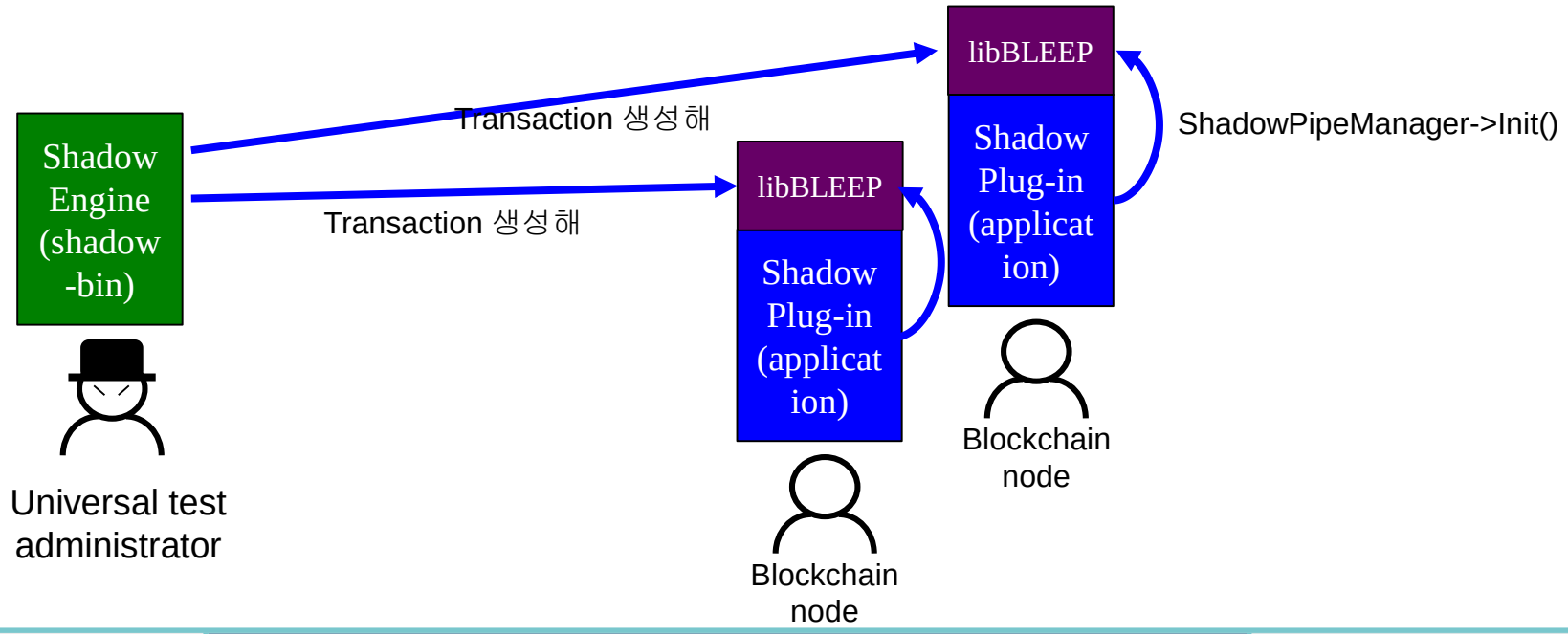
Implementing universal tests on BLEEP library

- 이를 위해 우리는 **shadow engine** 이 각각의 **node** 에게 **event** 를 전달할 수 있는 **interface** 를 구현하였다 **(shadowPipe)**
- 현재 버젼에서는 이 **interface** 를 사용하기 위해서는 각 **node** 에서 **shadowPipe** 를 생성하고 초기화를 해주어야 한다
  - 이는 shadowPipeManager class 의 Init 함수로 구현되어 있으며, 현재의 구현 템플릿에서는 state machine 을 초기화할 때, 이 함수를 호출하도록 구현되어있다



Implementing universal tests on BLEEP library

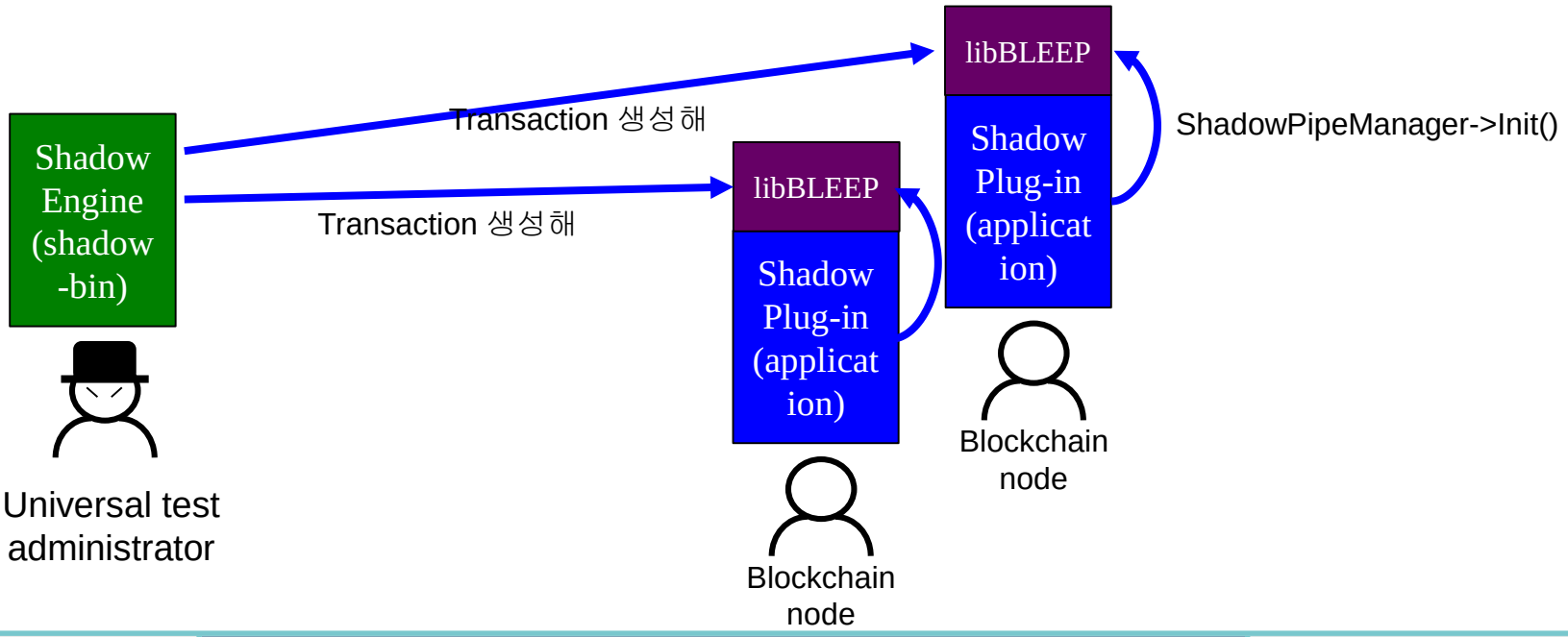
- 이렇게 **shadowPipe** 가 초기화된 **node** 들에게, **shadow engine** 은 임의의 **event** 를 원하는 시점에 전달할 수 있다
- 현재의 **event** 는 단지 **string** 을 **pipe** 를 통해 전달하도록 구현되어 있을 뿐이며, 따라서 각각의 **node** 에서 **string** 을 파싱하여 특정 행동을 수행하도록 구현해야만 하는 상황이다



ShadowPipe 를 통한 event 전달 예제

- **Xml** 설정 파일로, 각 노드에 전달할 **command, argument,** 그리고 전달하는 정확한 시점을 설정할 수 있다

```
<node id="bleep1">
  <application plugin="PEER" time="13" arguments="" />
  <command id="generateTx" starttime="30" arguments="1 2 30" />
  <command id="generateTx" starttime="30" arguments="1 2 30" />
  <command id="generateTx" starttime="30" arguments="1 2 30" />
  <command id="generateTx" starttime="30" arguments="1 2 30" />
  <command id="generateTx" starttime="30" arguments="1 2 30" />
```



ShadowPipe 를 통한 event 파싱 예제

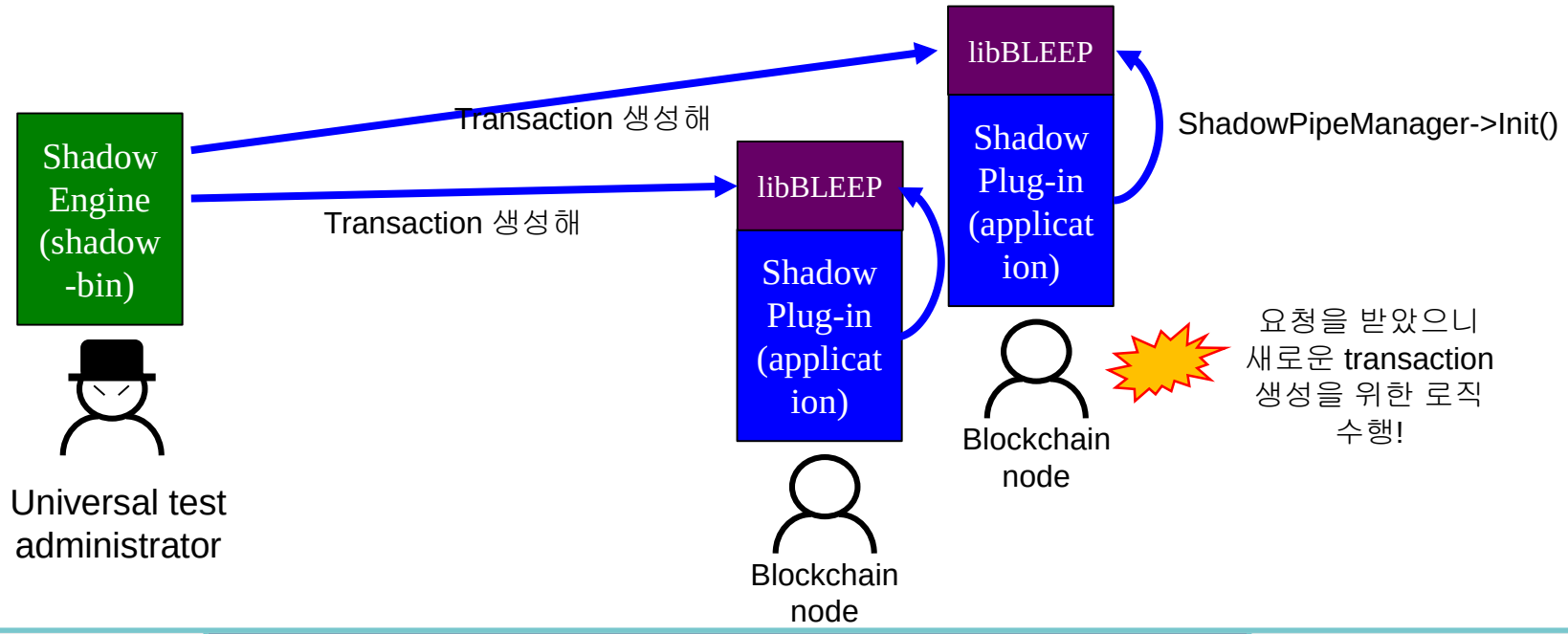
- 각 **node** 는 생성한 **shadowPipe** 로 **read** 가 가능할 때, **read** 를 수행하고, 이를 통해 받은 **command** 를 파싱하여, 특정 동작을 수행하게 된다

```
// check shadow pipe event
if (shadowPipeManager.IsEventTriggered()) {
    switch (shadowPipeManager.GetEventType()) {
    case ShadowPipeEventEnum::readEvent:
```

```
// receive command length
int length = 0;
n = read(fd,&length,sizeof(int));
```

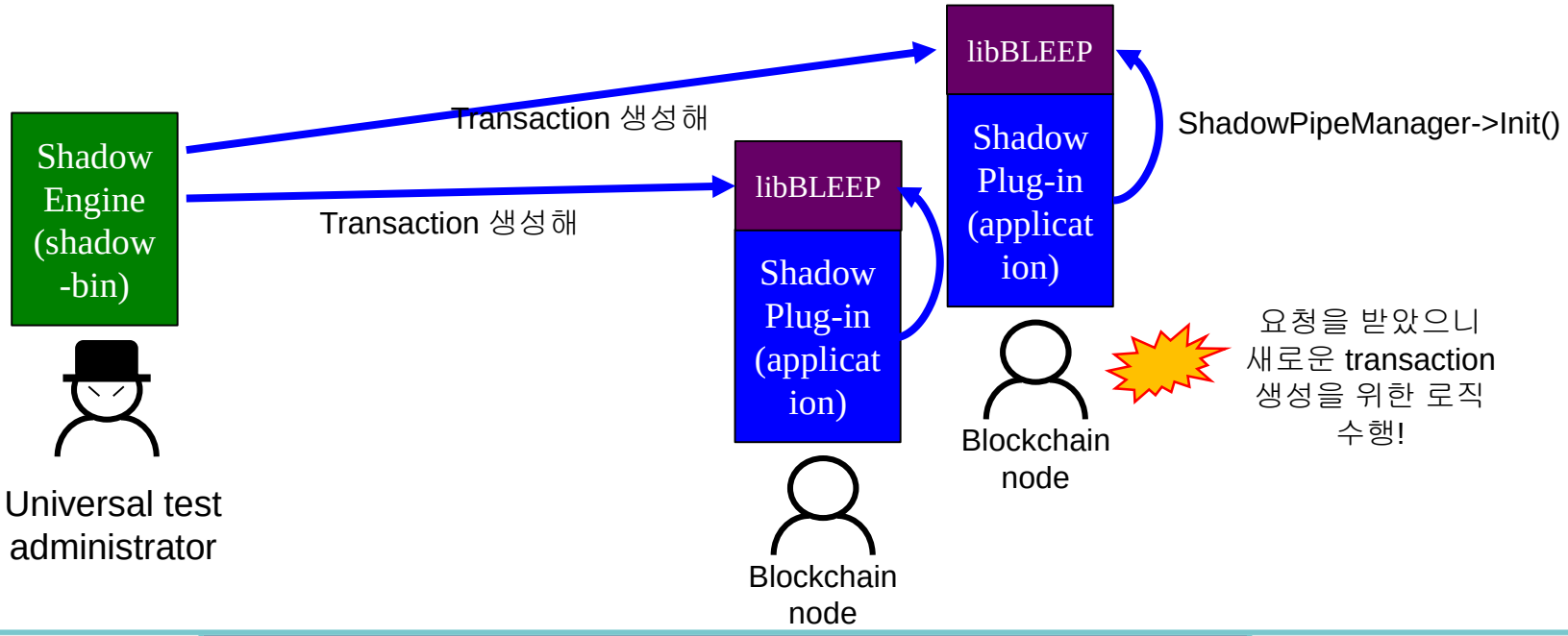
```
    if (cmd_id == "generateTx") {
        std::shared_ptr<Transaction> generatedTx(new SimpleTransacti
_args[1].c_str()), atof(cmd_args[2].c_str()));

        gStateMachine.txPool.AddTx(generatedTx);
        if (gStateMachine.txPool.GetPendingTxNum() >= block_tx_num)
            nextState = StateEnum::appendBlock;
    }
```



Implementing universal tests on BLEEP library

- 향후 **shadowPipe** 를 통해 들어오는 **command** 에 대한 **parsing** 은 **library** 에서 구현해줄 예정이며, **node** 개발자는 존재하는 **command** 에 대한 **handler** 만 구현하면 되는 형태로 제공할 예정
- 혹은, **handler** 를 구현할 필요도 없이, 각 **node** 에서 **shadow engine** 이 전달한 **event** 를 받았을 때, 자동화된 행동을 하는 기능을 **library** 내에 포함시킬 예정. **Ex)** 특정 **neighbor node** 에 대한 연결 **event,** 연결 중지 **event** 등



BLEEP : Blockchain testing library

- **BLEEP's main testing functionality**
  - Input control (transaction injection)
  - Protocol debugging (message passing)
  - Output verification (blockchain correctness check)

- **And many more in near future**
  - Configuration of node topology (Dynamic topology control & test)
  - Internet topology control
  - Various failure control (network failure, node failure, etc)

- **And 궁극적으로**
  - Common blockchain test benchmark suites

Thank you