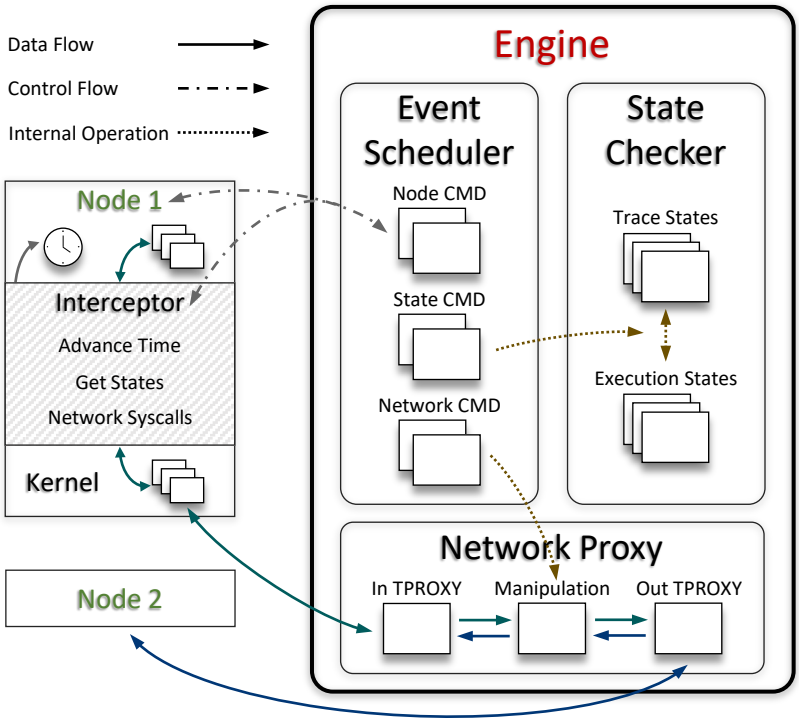
SANDTABLE 设计和说明文档

1. 设计目标

分布式系统中网络和节点存在不确定性，事件交互极为复杂，使得发现 bug、复现 bug、理解 bug 和修复 bug 存在巨大挑战。

针对以上困难和挑战，SandTable 技术使用形式化方法建模并探索分布式系统复杂的事件交互，使用代码操控技术确定性控制分布式系统中的不确定因素。操控技术对建模过程提供反馈并迭代修正建模错误，在无建模错误的情况下，模型发现的 bug 即为系统中真实存在的 bug，并可在操控技术的加持下确定性重现。模型检验可找到较短或最短触发路径，有利于理解 bug。待 bug 在模型中修复并验证后，再将此修复应用到代码中，可避免修复 bug 导致新的 bug 的情况。下图是 SandTable 技术的概述图。



本文将从协议层形式化建模和检验，以及代码层测试工具的设计和使用两方面展开。

2. 协议层形式化建模和检验

我们使用形式化方法进行协议层形式化建模和检验，形式化语言采用 TLA+，检验工具使用 TLC。下面将详细阐述建模过程和检验过程。

2.1. 协议建模过程

对分布式系统的形式化建模包括三个部分，代码中的核心事件建模、分布式系统环境建模和系统需满足的属性建模。

2.1.1. 代码核心事件建模

代码建模的主要目的是发现代码核心协议的 bug，因此，对于与协议无关的代码需要进行抽象。要抽象掉的部分包括但不限于日志输出、网络配置、文件读写、测试用例、上层应用等实现细节。抽象方式为保留核心变量，按事件粒度分模块建模。

模型的状态由变量的赋值组成，在完成对代码核心协议的抽象后，可以明确需要建模的关键变量。建模的过程围绕这些变量的当前状态和下一状态展开，它们组成了系统中的动作（或事件）。这个过程需要仔细阅读源代码，将代码中的变量赋值转译为 TLA+语言的变量赋值。TLA+严格要求一个动作需要对所有变量进行赋值（未改变的变量需要显示说明变量值未改变），否则 TLC 将给出警告或错误，这避免了某些因人的疏忽导致的建模错误，也要求开发者对协议有清晰的理解。

在 TLA+模型中，分布式系统的状态改变在离散的步骤中完成。这些离散步骤的粒度即为模型的事件的抽象粒度，我们建模时要求抽象粒度应对应于代码中相应事件触发的开始点和结束点。理论上，分布式系统的一个进程执行任意一行代码都可能遇到节点宕机的情况，但建模这种细粒度的规约需耗费大量人力，而模型检验产生的巨大状态空间则会包含大量无意义的状态。我们定义事件触发的开始点有：进程时间达到特定值、进程接收到消息。这些开始点都会触发系统执行特定函数，函数执行结束使得系统进入静止状态的位置即为结束点。在事件驱动的分布式系统中，这种事件抽象粒度符合直觉且容易建模。如果做更粗的抽象，例如将主节点选举过程的消息交互抽象掉并直接得到主节点，则难以对选主过程可能存在的 bug 进行探索。

建模过程可以分模块进行，并逐步组装。例如，Raft 基础协议可以将选主和日志复制作为两个模块。分模块建模可以降低整体建模复杂度，减少建模错误，增加对建模进度的宏观把控。

2.1.2. 分布式系统环境建模

分布式系统环境包括支撑系统正常运行的网络通信和时钟推进，以及影响系统正常执行的局部错误。

对于网络通信，真实分布式系统往往采用 TCP 或 UDP 协议作为传输层协议。因此，我们分别建模了这两种传输层协议的网络收发行为。TCP 协议是有连接且按序到达的，我们将一个连接中的网络包存储在二维序列的数据结构中，第一维表示发送者，第二维表示接收者。存储在该数据结构中则表示包已发出但未到达。取出连接中第一个包到接收者节点则表示包已到达。如果连接断开，则清空该连接的所有网络包。UDP 协议是无连接且无保证网络包的接收顺序和个数的，因此所有的网络包都存储于一个集合中，集合没有取出顺序，且可通过规则对其中元

素进行丢弃或重复。我们将这两种传输层协议封装为 TLA+库，以减少核心协议对网络操作的依赖。

在 TLA+中，一个行为包含从时间到状态的函数 F，F(t)表示系统在时间 t 的状态。我们建模一个系统即建模了系统可能的行为。因此，使用 TLA+建模分布式系统无需显式建模时钟。

分布式系统中存在网络错误和节点失效等局部错误情况。根据网络类型的不同，TCP 模型建模了网络分区错误。网络分区将节点划分为两个集合，集合内部可以相互通信，集合之间无法通信，在网络分区产生时，集合之间的消息全部被丢弃。UDP 模型建模了网络包乱序到达、丢失或重复。UDP 模型蕴含了网络分区错误，即两个分区之间的消息全部被丢弃。节点失效重启建模为一个事件使得失效重启节点的状态改变为重启后的状态。

2.1.3. 系统需满足的属性建模

分布式系统的属性有两种，安全属性（safety property）和活性属性（liveness property）。安全属性检查系统是否进入错误状态，活性属性检查系统是否能正常取得进展。这些属性是检验出系统 bug 的关键。

规约系统需满足的属性需要详细了解被建模系统的领域知识，例如，Raft 协议要求系统中不允许同时存在两个合法主节点。我们结合被建模协议基于的论文、代码和历史 issue，深入探究了这些系统需要满足的属性，并规约为安全属性和活性属性。

以 Raft 协议为例，其需要满足的安全属性有选举安全性、日志匹配、仅主节点可附加 log、提交的日志必须存在于多数派节点上等。其需要满足的活性属性有无主节点时最终能选出主节点、不断有日志项被提交等。

活性属性在检验过程中不能使用对称性优化、状态剪枝和仿真检验模式，使得在应用活性属性检验过程中存在诸多限制。我们主要关注于安全属性的规约和检验。

2.2. 协议检验过程

在系统建模完成后，通过模型检验可以检验出属性是否得到满足。然而，由于建模过程中的转译错误，模型可能失真，造成漏报和误报问题。由于状态爆炸，检验难以有效高效。我们通过模型覆盖率指标对模型进行初步 debug，通过与代码层工具获取到的状态信息进行对比，对模型进行深入细节的 debug，使得模型最终与代码一致。在此基础上检验出的属性违反可认定为真实存在于代码中的 bug。我们制定了一套参数来启发式剪枝模型状态，使得模型检验过程能够有效高效的探索到属性违反，且便于找到较短或最短触发 trace。

2.2.1. 模型覆盖率检验

TLC 工具可给出模型检验过程中的动作的触发次数，它可以作为最粗粒度的覆盖率检验指标来 debug 模型，如果存在被触发次数为 0 的动作，则大概率表明建模过程出现了问题。然而，这种检验指标粒度太粗，无法判断动作内部分支被触发情况。

我们在开发模型过程中，对动作各个分支打上一个 tag，存储于一个专门的变量中。使用随机仿真模式获取大量 trace，对 trace 的各个状态的 tag 进行统计，得到细粒度的模型分支覆盖率情况。通过这些覆盖率指标，可以得到触发次数为 0 的分支，这些分支需要被详细审查，以

确定它无法被触发的原因，此过程通常可以排除一些初级 bug。触发次数较少的分支往往是系统中难以测试到的边缘情况。这些信息将被进一步用于启发式参数设定。

2.2.2. 模型与代码状态对比

由于形式化语言与代码语言的天然差距，建模难免引入转译错误，这些错误难以被找出，使得检验结果存在漏报误报。我们通过代码层测试工具获取分布式系统中各个节点的状态，与模型状态进行对比，找出不一致的情况，即可消除转译错误。

与代码进行对比的 trace 可能违反或未违反协议属性，代码在执行过程中可能崩溃或不崩溃，对比结果可能为通过或不通过，对于它们的组合情况，我们可以分为以下几种情况进行讨论：

- 代码崩溃，触发了代码级 bug。由于代码崩溃条件可以认为是一种安全属性，因此即使 trace 未违反安全属性，也可通过添加上此安全属性使得它违反。对比结果如果不通过，需要进一步修正模型的不一致，但代码的崩溃原因仍然是一处 bug。
- 代码不崩溃
  - 比对不通过：这条 trace 与代码不匹配，则 trace 是否违反协议属性没有意义，需要进一步修正模型的不一致。
  - 比对通过：这条 trace 与代码匹配。
    - Trace 未违反安全属性：无特别用处，提升了模型建模与代码一致的信心。
    - Trace 违反安全属性：发现协议级 bug，且能在代码中重现。

通过状态对比一轮一轮的迭代，可以消除所有的转译错误，并最终无误报的发现代码级和协议级 bug。这些 bug 的发现可以用布尔表达式 (代码崩溃 || (比对通过 && trace 违反安全属性)) 表示。

2.2.3. 启发式状态剪枝

模型检验存在状态爆炸问题，使得大量状态无法在有限时间内被检验。我们提出一种启发式剪枝方式，通过给特定操作设定次数限制，引导 TLC 模型检验工具对事件的调度，从而更有效高效的发现属性违反。

我们在模型检验过程中发现了大量无意义的 trace，例如，网络分区发生得过于频繁，使得在仿真模式中，大多数 trace 难以有效选举出主节点。我们尝试过对各个动作设定随机触发概率，使得 1000 个状态量级的 trace 能成功选举一次主节点。但随机触发概率仍不够有效高效，且模型检验模式无法使用随机的特性。我们对各个操作设定被调度次数限制，使得模型检验工具在次数用尽时不得不调度其他可调度动作，并进一步探索他们的组合情况。

这些参数包括：网络发送次数/接收次数/缓存消息数量、丢包/重复包/乱序到达/网络分区次数、节点失效次数、选举次数、心跳发送次数等等。

根据短时间的仿真模式得到的反馈，例如分支覆盖率、事件触发种类和模型深度等信息，再结合人的经验，可以对这些参数赋一个合理值，使得系统的边缘情况能够被频繁触发。

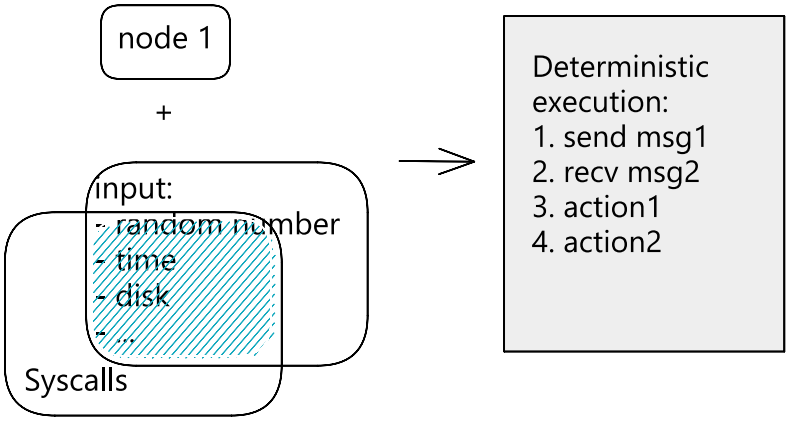
由于活性属性的检验方式与安全属性不同，使用启发式状态剪枝参数会使得活性属性检验失效。因此，规模较大的模型难以有效检验活性属性。

3. 代码层测试工具的设计和使用

分布式系统的确定性操控技术可使能模型与代码的状态对比，对于订正模型、发现代码级 bug、确认协议级 bug、重现 bug、理解 bug 和修复 bug 有重要作用。我们的代码层测试工具使用透明网络代理技术进行网络截获和动态符号解析技术进行系统调用截获，透明地控制了分布式系统中绝大多数不确定性，使得系统被我们的工具操控执行。模型 trace 的事件被转换为控制器测试输入，指导分布式系统执行指定 trace。下面将详细阐述控制器、截获器，以及配置和运行脚本的设计和使用方式。

3.1. 截获器

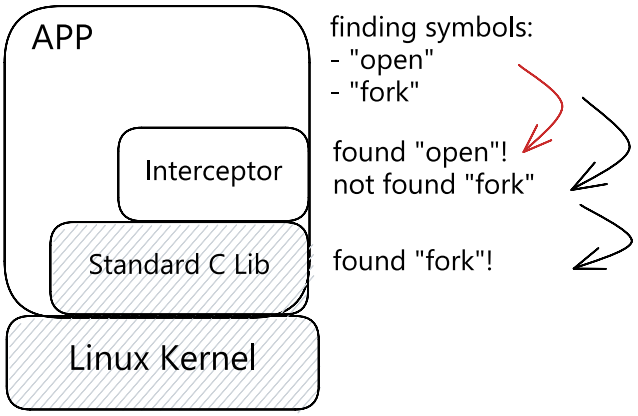
截获器需确定性控制单节点中受控程序的执行。影响程序不确定的因素包括，节点失效、网络错误、时钟事件、随机数等。我们可以将这些因素当作程序特殊的输入（或环境输入），如果对这些输入完全控制，则程序可以确定性执行我们给定的 trace。系统调用是程序获取这些输入的主要方式，例如，程序获取随机数需要先设定种子，通常选取当前系统时间作为种子，如果截获了系统时间，则可确定程序获取到的随机数。因此，截获器的一个主要任务是截获系统调用。下图为程序确定性执行的输入的关系。



3.1.1. 截获器技术选型

有多种技术可以对系统调用进行截获，如 LD_PRELOAD、ptrace()，和 SecComp/BPF 等[1]。我们使用了 LD_PRELOAD[2]基于动态符号解析的截获技术。下面将对比这几种技术的优点和缺点。

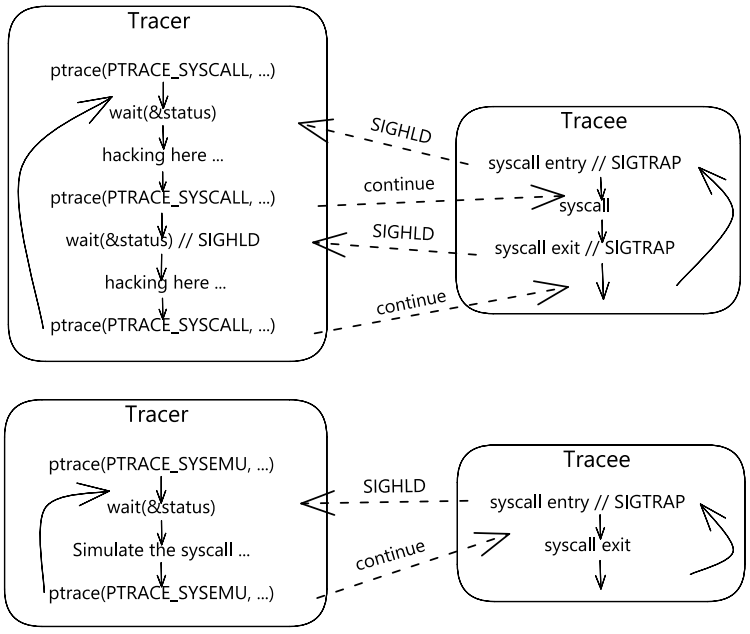
LD_PRELOAD 是一个环境变量，通过它设置一个预加载的动态链接库，可影响链接器进行符号解析的顺序，使得预加载库中的符号地址被优先解析到。通过这种方式可以覆盖掉 C 标准库或其他动态链接库中的符号地址。程序通常不会直接使用汇编语句 INT 0x80 或 SYSCALL 进入内核执行系统调用，而是通过调用 C 标准库中的系统调用封装函数来进行系统调用。系统调用封装函数给系统调用提供了标准的入口，是 C 标准库中的普通函数，可以在符号解析过程中被预加载库中的符号替代。下图为 LD_PRELOAD 技术工作原理的示意图。



此外，预加载库还可以定义先于受控程序 main 函数执行的函数，因此可以在程序执行前初始化配置和网络连接等。

LD_PRELOAD 的优点有：简单、易用、通用、高效；运行于受控进程地址空间，可直接访问和修改进程资源等。缺点有：仅可用于动态链接程序，仅可用于使用了 C 标准库的程序（在 Linux 环境中，Go 语言不使用 C 标准库，无法对使用 Go 语言编写的程序进行截获[3]，但可在 OpenBSD 中进行截获[4]），在无法设置 LD_PRELOAD 环境变量的情况下不可使用等。

Prace()[5]是 Linux 的一个系统调用，主要用来调试和分析进程的执行。我们将调试器进程称为 tracer，被调试进程称为 tracee。Tracer 对 tracee 有完全的控制权，可以读取和设置 tracee 的 CPU 寄存器值，读取和设置 tracee 内存地址值。而强大的能力意味着对细节的控制较为复杂。对于我们关注的截获系统调用的情况，ptrace 提供 PTRACE_SYSCALL 和 PTRACE_SYSEMU 选项对系统调用进行截获或模拟。下图展示了使用 PTRACE_SYSCALL 和 PTRACE_SYSEMU 进行系统调用截获或模拟的过程[6]。



Prace()的优点有，可以截获任意程序，可以精确控制程序执行。缺点主要为使用复杂，需要较深入的专业知识；运行开销很大，如上图一次系统调用截获将引入 4 次上下文切换；ptrace 控制的程序不方便结合其他动态分析技术（如 valgrind）使用；对于一些特殊的不进入内核执行的系统调用（vDSO[7]），需要特殊处理（即关闭 vDSO）。

SecComp 技术[8]关注于安全性，可以限制进程执行一个系统调用子集，即 filter 掉不允许执行的系统调用或返回给定结果，多用于沙盒和容器技术。BPF 可以在 Linux 内核虚拟机安全运行一段 BPF 程序，多用于 trace/filter/monitor 网络和系统功能。BPF 通常与 SeComp 技术结合使用，用于实现 SecComp 的 filter。这种系统调用截获技术安全高效，但面临一些限制，如无法修改系统调用参数等。可进一步结合 ptrace，通过设置 SECCOMP_RET_TRACE 参数返回到 tracer 进程，让 tracer 进程对 tracee 进行修改。

我们最终采用了 LD_PRELOAD 技术，其简单易用性和较大的适用范围可以表明 SandTable 方法的有效性。

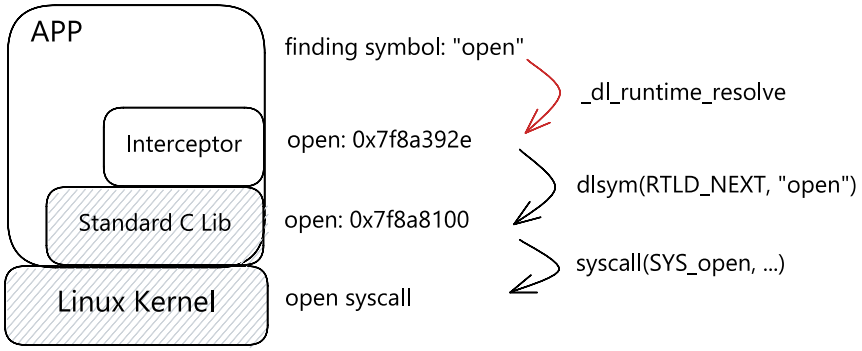
3.1.2. 截获器框架

截获器框架需要处理符号解析任务和与控制器连接并执行控制器命令的任务。

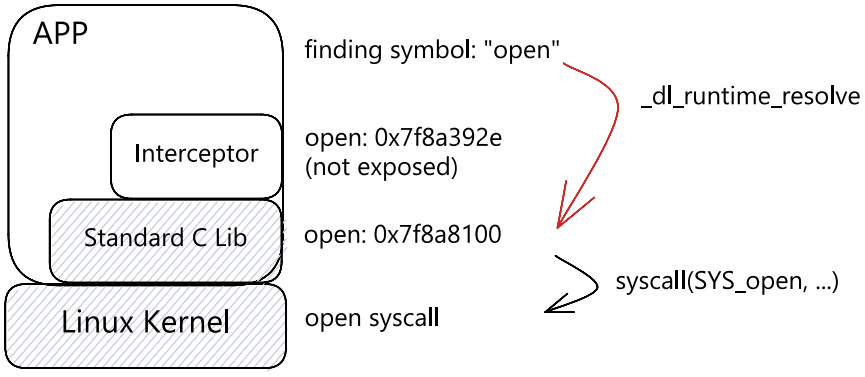
对于符号解析任务，我们需要覆盖 C 语言中相应符号，并获取被覆盖符号的地址使得截获器可以执行真实系统调用。覆盖符号只需要将符号名称定义为 C 语言中相同即可。获取被覆盖符号的地址需要使用动态连接器 ld.so 提供的 dlsym(void *handle, const char *symbol)函数[9]，该函数的 handle 参数除了可以使用动态链接库之外，还有两个特殊取值：

- RTLD_DEFAULT：第一次出现（interceptor 的符号）
- RTLD_NEXT：下一次出现（libc 的符号，真实地址）

通过 RTLD_NEXT，我们可以可以查询到被覆盖符号在 libc 中的真实地址。下图为符号解析到真实系统调用过程示意图。



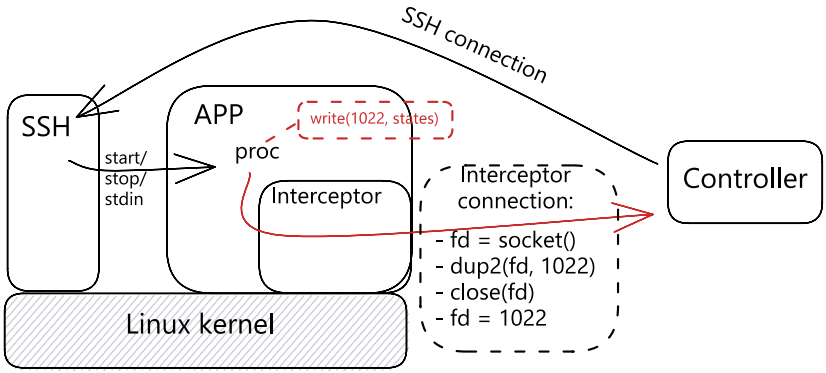
在编译链接过程中，可以设置链接选项设定 version script 来控制符号的可见性，使得我们可以控制是否截获特定的系统调用。此外，截获器还提供运行时 enable/disable 特定系统调用的截获功能。下图为使用 version script 控制了 open 符号可见性后的截获流程示意图。



对于与控制器建立连接的任务，截获器通过 socket 连接至控制器监听的 TPROXY 端口（TPROXY 监听的 socket 仍可当做普通 socket 使用），控制器判断该连接的目的地址为控制器本身，将该连接分类为与截获器建立的连接。这条连接的作用有两个：

- 接收并执行控制器对截获器的命令。例如，推进系统时钟。
- 对控制器在节点执行的命令进行 ack，ack 可携带数据，使得控制器确保这条命令执行完毕且可进行下一个命令的执行，或获取到节点的状态信息。

为了使被控程序可以使用到这个连接，截获器通过 dup 系统调用将该连接的文件描述符设置为特定值（例如，设置为 1022），下图为该 socket 文件描述符的设置过程。



3.1.3. 系统调用截获

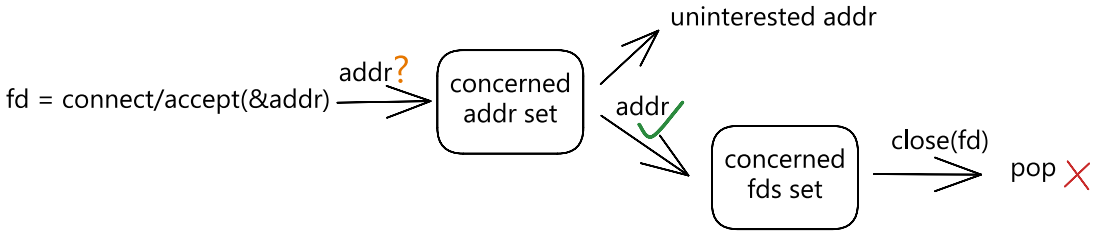
我们目前主要截获网络相关系统调用、时间相关系统调用和部分文件相关系统调用。

对于网络截获，由于控制器 TPROXY 网络代理技术已处理大部分任务，使得网络截获在系统调用层面需要做的事情并不多。主要任务包括：

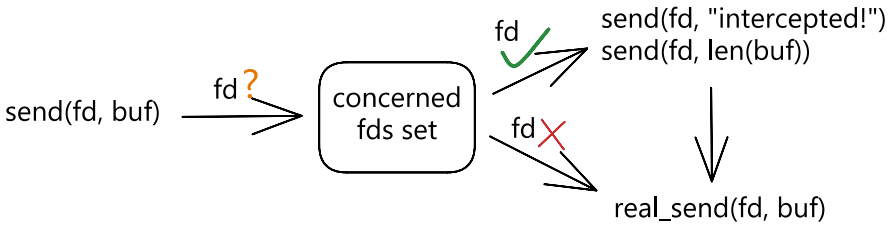
- 区分该连接或消息是否被截获，使得控制器截获或放行网络包。
- 标识 TCP 消息边界，使得控制器正确对接收的 TCP 数据流分段为一个个消息。
- 监控被截获的 socket，用于检查连接状态和接收队列是否有到来消息。

因此，我们需要截获以下三种网络相关系统调用：

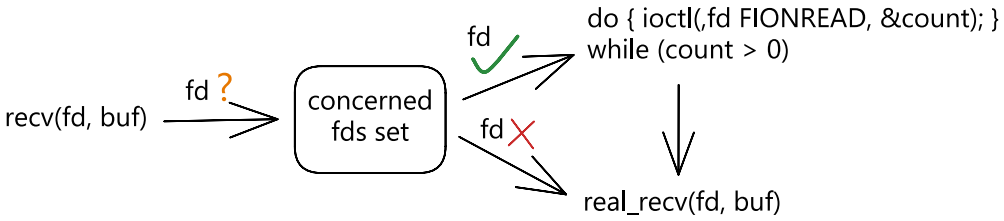
- 与建立/关闭连接相关的系统调用：connect/accept/close 等。以 TCP socket 为例，当截获到 connect/accept 时，检查连接对段的地址是否是配置中关注的地址，如果是则将该连接的文件描述符加入关注集合。在关注集合中的文件描述符在关闭时会从中删除。截获后处理方式如下图所示。



- 与发送数据相关的系统调用：send/write 等。如果 send 相关系统调用的文件描述符在关注集合中，则在发送实际数据之前，先发送一个 header，header 中包含一段特殊标识说明这一段数据是被截获的，还包含数据长度信息。截获后处理方式如下图所示。



- 与接收数据相关的系统调用：recv/read 等。接收数据时，如果接收队列为空，进程则会阻塞，导致后续命令无法得到执行。因此在实际接收之前需要检查接收队列，当接收队列不为空时才调用接收相关的系统调用。截获后处理方式如下图所示。在实际使用中，我们进行了简化，先使用控制器判断节点是否可接收消息，再调度节点消息接收事件，因此，可以关闭对 recv 系统调用的截获。

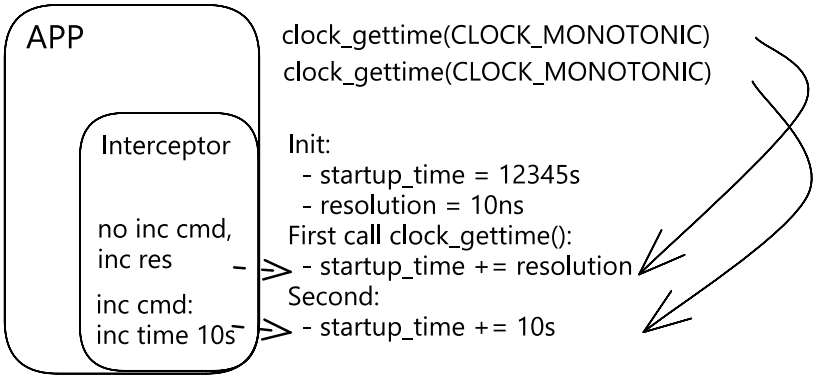


这些系统调用有多种变种版本，使用方式多样，需要较多的特殊处理。例如 accept 有 accept4 系统调用变种，connect 使用非阻塞 socket 时直接返回-1（EINPROGRESS）等。

对于时间的截获，主要任务为推进进程的时钟以触发特定事件，在系统调用层面分为三类与时间相关的系统调用：

- 获取时间：如 time/gettimeofday/clockgettime 等系统调用。
- 异步事件执行（timer）：如 alarm/timer_create 等系统调用。
- 设置超时的阻塞等待：sleep/select/epoll 等系统调用。

第一种与获取时间相关的系统调用是进程观测时间的基础，截获器实现了一套时间控制机制，模拟系统真实时间和进程执行时间。在截获器初始化时，设置一个特定的启动时间，如 2022 年 1 月 1 日。当受控进程调用 time/gettimeofday/clockgettime 时，截获器不调用系统调用获取时间，而是返回截获器中存储的时间，并推进一个较小的 resolution 时间来满足系统时间是单调递增的属性。当截获器收到来自控制器的推进时间命令时，将时间推进给定值。下图为截获与时间相关系统调用的示意图。



我们分析了多个程序，发现他们对时间的使用方式相似，先设置一个截止时间，在一段时间后，再次获取时间并与截止时间对比，如果达到截止时间，则触发相应事件执行。伪代码如下：

```
start_time = clock_gettime(CLOCK_MONOTONIC);
deadline = start_time + timeout;
// ...
current_time = clock_gettime(CLOCK_MONOTONIC);
if (current_time > deadline) {
    // do something...
}
```

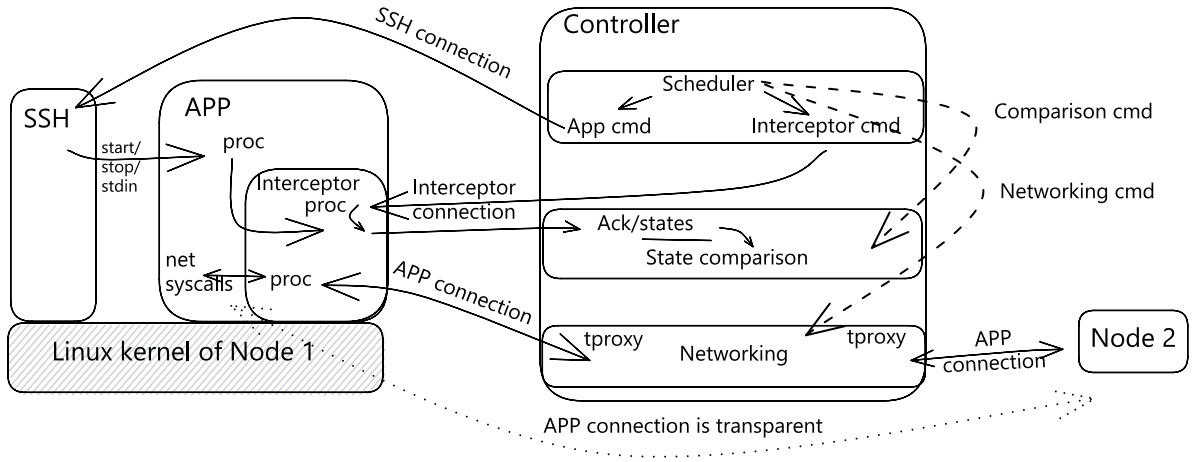
对于这种使用时间的方式，即使事件被尚未截获的异步 timer 触发或被带有 timeout 的 epoll 触发，程序也能按照预想的执行路径执行。

我们暂未实现对 timer 和带有超时的阻塞等待系统调用进行截获。上述对获取时间的截获可以覆盖大量使用场景，timer 和阻塞等待系统调用仅影响执行效率，如果将它们的超时设置得足够短，则执行效率也可以得到提升。

对文件相关的系统调用截获，目前截获了 open/read/write/close 等系统调用。由于每条 trace 的控制执行都是头开始，所有文件都为初始化状态，因此暂不需要对文件操作进行特殊处理。

3.2. 控制器

控制器负责事件调度、网络截获和状态收集，运行于一个独立节点，所有网络流量都经此节点进行路由。下图为控制器工作原理简图。



3.2.1. 事件调度

控制器接收命令来对全局事件进行调度控制。命令输入来源为命令行输入或文件输入，命令类型分为受控节点执行的命令、截获器执行的命令、网络命令和状态比对命令。

命令行输入模式允许开发者使用控制器内置命令交互式地手动控制分布式系统执行，常用于调试和理解分布式系统执行。文件输入通常来源于模型 trace 转换后的事件调度输入，使得控制器可以批量对 trace 进行确定性重放执行。目前事件调度尚不支持全自动随机事件调度。

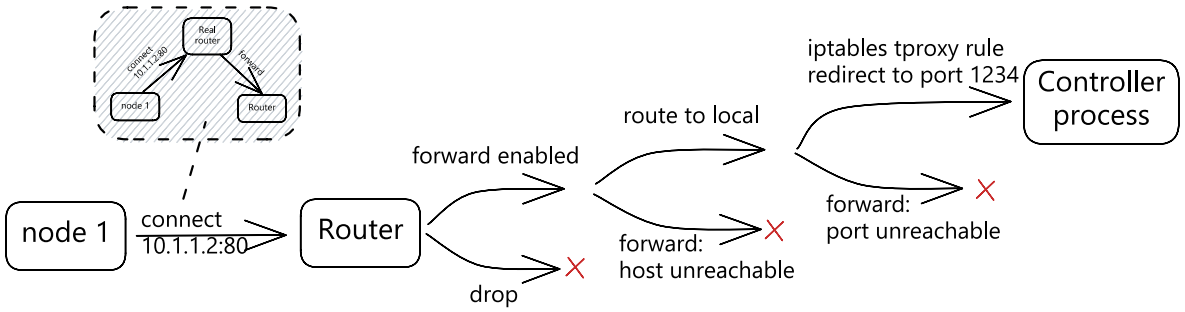
控制器支持的命令有以下四种：

- 受控节点执行的命令：此类命令通过 SSH 连接发送到受控节点，在受控节点中开启一个虚拟终端，虚拟终端接收到的输入会被传入 shell 或受控程序，这取决于终端前台运行的程序是哪个。因此控制器可以发送启动命令使得受控程序启动，发送终端信号快捷键（如 Ctrl+C）控制程序运行状态，发送程序可接收的命令（封装了公共 API 的函数）执行指定代码事件。
- 截获器执行的命令：截获器是受控程序和内核的连接层，因此，可以控制系统调用和获取受控程序数据。例如，执行推进系统时间命令、检查 socket 是否可接受数据等。
- 网络命令：网络截获集成在控制器内部，网络命令直接在控制器中执行，可以发送网络消息给节点、注入网络错误等。
- 状态比对命令：状态比对命令使得控制器对特定节点的 trace 状态与代码状态进行比对，如果比对失败，控制器抛出异常。代码状态数据来自于发送受控节点执行的命令到节点，命令包含获取给定状态的函数调用。

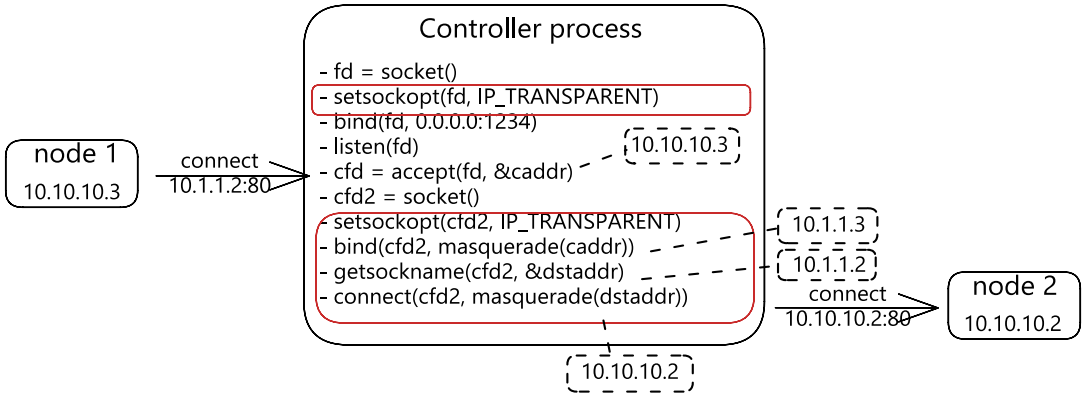
3.2.2. 网络包控制

我们使用了一种 Linux 内核特性 TPROXY 透明代理来获取网络包。这项技术包含三个部分：配置转发规则、监听透明代理端口和使用透明代理连接到对端。完成对网络的代理后，将收到的网络包缓存，并根据调度器命令执行，可达到对网络的确定性控制。

配置转发规则使得网络包正确路由到控制器所在节点。需要开启内核路由转发，添加路由规则将特定子网的网络包路由到本地，添加防火墙 TPROXY 规则将该子网网络包转发到指定端口。通过这些配置，网络包可以从节点之间正确路由到控制器。下图为配置的流程图，为了使用 docker/lxc 等工具自动化测试，实际路由转发过程更加复杂（如虚线框所示）。

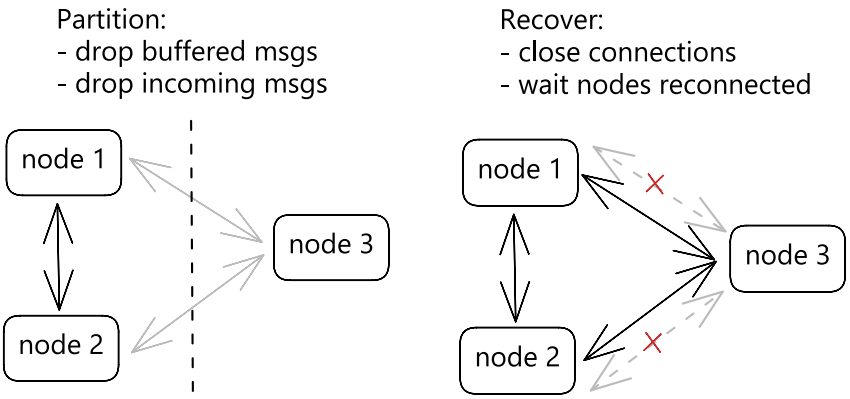


监听透明代理端口需使用 IP_TRANSPARENT 参数设置的 socket，去监听防火墙转发规则中设置的端口（图片示例中使用了 1234 端口），这样使得被代理的网络包能被控制器接收。再使用一个 IP_TRANSPARENT 参数设置的 socket，绑定到发送者的 IP 地址，并发送给接收者，使得接收者看到的发送者的 IP 地址仍然是原发送者，而不是控制器。下图描述了对 TCP 网络截获的关键代码，虚线框中标识了相应代码中使用的 IP 地址，在本例中，应用认为他们在 10.1.1.0/24 的虚拟子网中建立了连接。

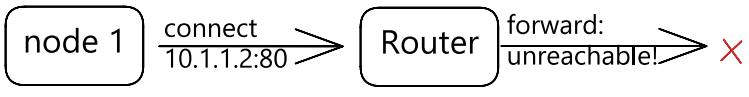


上述技术完成了对网络包的截获，被截获的网络包缓存在控制器中。根据网络传输层协议的特点，TCP 协议的网络包按连接存储于 queue 中，例如 deqeue(msgs[src][dst])即可获取 src 发送到 dst 连接中第一个网络包，UDP 协议的网络包存储于从序号到网络包的映射表中，例如 msgs[0]即可获取控制器接收到的第一条消息。对这些网络包数据结构进行控制，即可控制网络的发送、丢包、重复包、分区等情况。

在 TCP 网络分区的模拟中，控制器不会立即断开连接，而是在分区恢复时断开连接，并通过截获器命令推进节点时间，触发节点网络重连。这样使得节点在分区时观察到连接中无消息传来，在分区恢复时重建连接。下图为网络分区模拟的示意图。

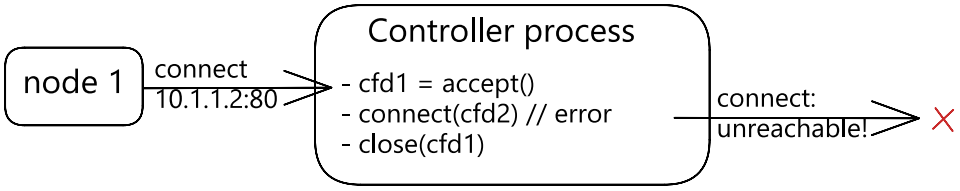


网络透明截获技术在大多数情况下保持代码原有行为，但存在某些情况下会改变代码原有行为。如下图所示，在 TCP 网络连接失败的情况下，node 1 能正确得知失败原因。



Node 1 gets error msg: no route to host / connection refused

而在控制器的代理下，node 1 观察到连接成功又被迅速关闭。



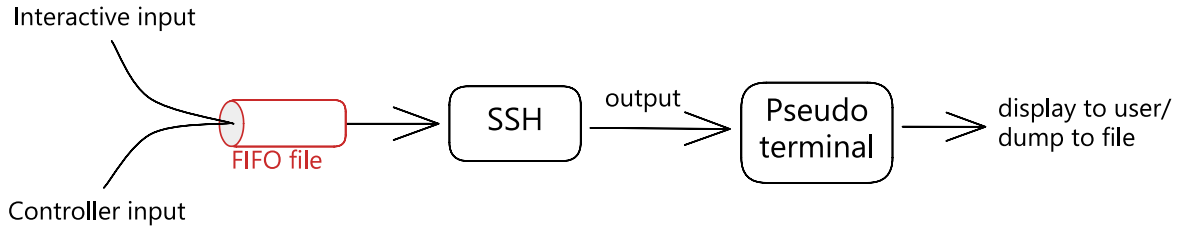
Node 1 thinks the connection is quickly closed after establishment.
Controller knows the peer is unrecheable.

尽管可以通过复杂的防火墙规则来使得该行为仍与原行为保持一致，但在本工具主要用于测试协议级 bug 的使用场景下，这种情况暂未进行处理。

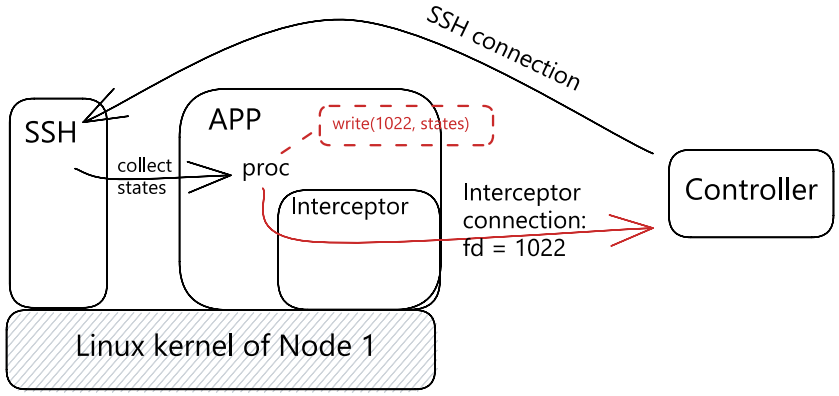
3.2.3. 状态收集

状态收集用于模型与代码状态比对，以发现模型转译错误，或确认模型层发现的 bug。

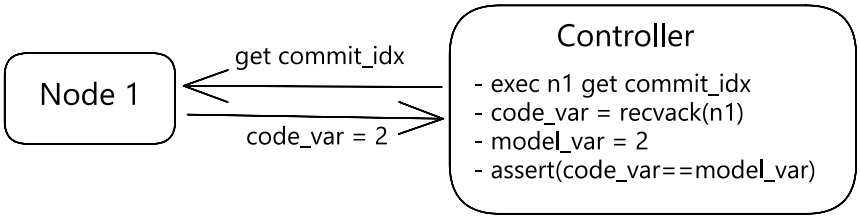
该技术通过两种方式收集代码状态。一种方式是通过正则表达式处理代码的 debug 日志输出，并提取重要的变量值。另一种方式通过 SSH 连接通道发送特定状态收集命令到受控程序，并通过截获器返回结果的连接收集返回数据。之所以不使用 SSH 连接直接返回数据，是因为 SSH 连接被设计为可交互的，标准输出产生大量难以有效处理的数据到虚拟终端，这些结果可以交互式的展现给开发者进行调试，但控制器难以直接使用。因此 SSH 连接通道是单向的。下图为 SSH 连接的示意图。



截获器与控制器建立的连接可以在受控程序中使用，因此，返回结果数据时，只需要直接写入该连接的文件描述符即可。下图为控制器发起状态收集，到得到返回结果的过程。



收集到节点状态信息后，使用 2.2.2 模型与代码状态对比技术完成状态对比和 bug 分析工作。下图为一个变量状态的比对过程。



参考文献

[1] Timmmm, “Answer to ‘how could I intercept linux sys calls?,’” *Stack Overflow*, Nov. 05, 2021. https://stackoverflow.com/a/69852132/9543140 (accessed May 29, 2023).

[2] “ld.so(8) - Linux manual page.” https://man7.org/linux/man-pages/man8/ld.so.8.html (accessed May 29, 2023).

[3] “Go doesn’t *need* to bypass libc on Linux. There’s no technical reason to do tha... | Hacker News.” https://news.ycombinator.com/item?id=20855829 (accessed May 29, 2023).

[4] “OpenBSD system-call-origin verification [LWN.net].” https://lwn.net/Articles/806776/ (accessed May 29, 2023).

[5] “ptrace,” *Wikipedia*. Mar. 15, 2023. Accessed: May 29, 2023. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Ptrace&oldid=1144844815

[6] “Intercepting and Emulating Linux System Calls with Ptrace.” https://nullprogram.com/blog/2018/06/23/ (accessed May 29, 2023).

[7] “vDSO,” *Wikipedia*. Mar. 25, 2023. Accessed: May 29, 2023. [Online]. Available: https://en.wikipedia.org/w/index.php?title=VDSO&oldid=1146593908

[8] “seccomp,” *Wikipedia*. May 06, 2023. Accessed: May 29, 2023. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Seccomp&oldid=1153384375

[9] “dlsym(3) - Linux manual page.” https://man7.org/linux/man-pages/man3/dlsym.3.html (accessed May 29, 2023).