阅读报告

mTCP

mTCP 来自 NSDI'14，是 KAIST 大学写的关于提升多核系统中 TCP 性能的问题。现成的 Linux TCP 协议栈对这种短连接主要存在几个问题：

1. Connection locality：监听连接的常常是同一个 socket，各种线程会竞争，因此需要加锁；执行 TCP 连接代码的和调用 socket 发送接收数据的代码可能不在同一个核上，因此 cache 缺失率很高。很多解决方案解决了各种问题。
2. Shared file descriptor space：新的连接需要申请新的 fp，而操作系统中 fd 的分配是寻找最小可用 fd，这需要锁来保证互斥操作。
3. Inefficient packet processing：每个包的处理会导致内存分配和 DMA 的效率不高。引入 batch processing 解决，但没有完全支持 TCP 协议栈，也没有都被 kernel 所吸纳。
4. Heavy system call overhead：对于应用而言，每次调用 socket API 都需要使用系统调用，在用户态和内核态进行切换，这会造成很大的时间开销，浪费 CPU 资源。虽然已有一些批处理或系统调用调度算法，但用户或系统调用一旦发生改变，就需要修改用户应用或内核代码，显得十分冗余。

并不直接修改内核代码，而是从用户态为应用提供类 socket 的编程接口，这样应用移植性比较好。原因是

1. 建立用户态的 TCP 协议栈，要把那些优化合并进去要比改内核容易
2. 可以直接利用快速的 packet IO 工具(DPDK)
3. 更方便支持批处理

mTCP 还做了一些改进：

1. 使用多核可共享、但无锁的数据结构，单一生产者、单一消费者队列，不需要锁；
2. 使用 RSS 进行负载均衡，并为每个应用 thread 在同个 CPU 核配置 mTCP 线程；
3. 针对短连接：
    a. 尽可能优先收发控制类型的包，滞后数据报文；
    b. 针对要对每个新的 socket 申请新内存或之后释放空间的问题，我们采用预先分配大内存，并由 mTCP 来管理。

问题是

1. 原本内核中实现的许多网络协议/模块，尤其是一些安全检测模块，需要被重新实现，而且实现过程如何保证正确性是个问题；
2. 内存空间的管理。由于每次申请大内存，我怀疑这些内存无法被高效地使用；
3. 从下图可以看出，为了减少进程切换次数，设置了 buffer，这造成数据在多个 buffer 区域之间被来回传输，这些 copy 操作会造成 CPU 时间消耗；虽然用户态 CPU 时间多了，但还需要考虑是不是被浪费了。

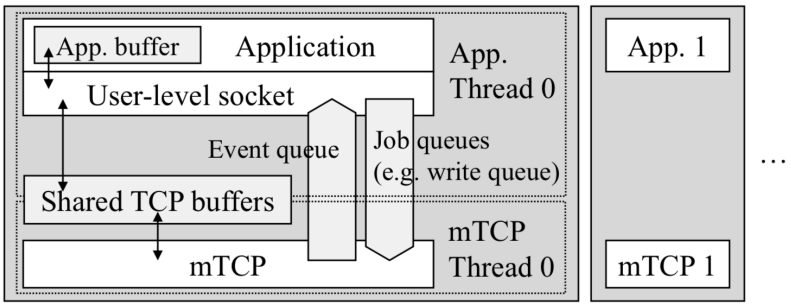


Figure 4: Thread model of mTCP.

Fastsocket

Fastsocket 解决的是 TCP 短连接存在较多互斥锁竞争的问题。与 mTCP 不同的是，Fastsocket 不是自建用户态的 TCP 协议栈，而是维护了原有的协议栈的功能，这样能够保证原有的大量功能仍然能够被使用；去模仿实现协议栈，需要较大且冗余的开发过程；同时自建协议栈可能没有完全符合 RFC 规定；还有 NIC 一旦与用户态协议栈绑定，就再也无法使用内核的协议栈。（DPDK 也是这样的）此外，提供新的 API 还需要修改应用程序代码。综上，Fastsocket 没有轻易放弃原有的 BSD socket API。

Fastsocket 需要解决的核心问题和对策：

1. 多核平台的 TCB 是全局的，那么访问时需要锁操作
2. 全局的监听表 listen socket，这要求所有连接都经过同一个 socket 队列——建立 local listen table
3. 全局的 socket 表（找最小的 fd）,可以减少锁的粒度，但不是个好办法——建立局部表

FastSocket 通过 hash 函数的方法，将 port 域 hash 到 cpu 核的 id。对于主动连接（由运行 FastSocket 的 host 发出），fastsocket 可以选择一个合适的连接源端口，从而使得 hash(port)=core id，之后收到包时检查目的端口，进行 hash，则可以保证局部性；对于非主动连接的包，如果采用上述方式，可能会将包交给并没有能接受这个流应用所在的 core。因此 fastsocket 区分了这两种连接，并映射到正确的 core 上。

FastSocket 还改进了 VFS，使得 socket 成为轻量级的文件描述符，跳过了许多初始化创建和销毁操作。（但其实没有改变 fd 的分配规则，也就是每来新的 SYN，创立新连接时，仍然会搜索最小的 fd）

FastSocket 还可能存在局限性，文中所用的 RDD+hash 具有随机性，较难保证大小流的均衡。考虑到是短连接，或许可以

THE

Thoughts on THE

THE 是早于 Unix 系统的由大牛 Dijkstra 开发实现的多道程序系统。其在 Introduction 里并不介绍自己的系统，而是强调做科研要讲究时效，传递科研要全心全意投入、这方面科研需要最聪明的人、经验不代表智慧等观点。在硬件资源有限的情况下，对于多道程序系统，其核心是要完成对于硬件资源的分配利用。

1. CPU 采用分时机制，轮流服务多个程序，实现了时间中断，而保证程序不会独占 CPU 很长时间；当 CPU 分时，每道程序并行执行时，可能会使某些数据被同时修改

或读取。THE 提出了信号量以及 PV 操作来保障同步资源的互斥问题，使程序不必一次执行完。信号量机制直到现在仍然广泛使用。

2. 内存不够用情况下，要保证当前程序所要使用的数据能在内存中，当不能全部放入内存时，大的程序或数据需要分段（segment）。同时把存储分为大小相等的 pages，每个 segment 可以放到一个 page 中。保存 segment 标识符来记录 segment 所在的位置。这样做到第一存储和第二存储的联合使用。其实，THE 的存储管理是虚拟存储的影子，虽然没有标准化的页表机制来实现虚拟地址与物理地址的隔离，但是分段标识符能将 segment 地址与 page 地址进行映射，已经实现了初级的虚拟内存。

THE 将 OS 系统进行层次化开发，从软件工程角度认为这样有利于进行系统的正确性测试（将系统切成多个模块，只需要对每个模块进行“单元测试”）。从最底层的硬件资源，自底向上分别是 CPU 调度、内存调度、控制台控制、设备控制、用户程序、操作人员。

以现在的观点来评论 THE，可能会发现以下问题：

1. THE 实现了多道程序（进程）同时运行，也实现了进程间竞争的问题，但没有实现进程间通信，可能当时的任务是相对独立的，没有这方面的需求；
2. 层次之间的依赖不是十分必要，特别是控制台层与设备层的关系，这两层是否可以合并，层次过多是否会带来效率上的损失；

总之，THE 是操作系统演进过程中的雏形，在 CPU、存储管理等上算法不如今天的系统完善，但实现单道到多道系统的跨越，解决了进程同步互斥的问题。

SafeBricks

SafeBricks 是基于 NetBricks 平台，利用 Intel SGX 和一些加密技术加强 NF 的安全；目标是针对公有云中部署 NF 时，可以保障 NF 以及 NF 处理的流量的安全性。

之前有一些工作包括使用加密技术来保障 NF，但使用密码学技术的后果是使性能大幅度地下降（复杂加密和解密消耗大量的 CPU 时间），而且采用这些方法的平台只能简单做一些+或<的计算，会使功能受限；还有一些方法会使用权限管理方法（比如，mcTLS）对硬件所能够接触的数据包的域进行监控和限制，但对于整个云而言，所有 NF 都部署于其之中，也就是控制所有节点，也能获得数据包。

SafeBricks 使用 Intel SGX 中的 enclave 将 NF 包裹起来，这种硬件手段可以阻止被入侵的操作系统或者有 root 权限的管理员获得被包裹的数据。而可以想象，这种将 NF 运行程序与操作系统隔离开来会导致进行系统调用时（收发包），也就是受信任的程序与不受信任的交互会造成较大的开销。针对这一点，SafeBricks 改进了 IO 方法，分配共享内存，设立缓冲队列，被包裹和未被包裹的代码通过队列完成交互。

注意到 SafeBricks 特别强调 One Core，也就是其将 NF 尽量放在一个 core 上，这样减少产生 NF chain 时，不同 NF 之间传递报文会涉及到加解密过程，使性能下降。SafeBricks 继承了 NetBricks 的方法，将 NF 当成一个个处理函数，使用 Rust 语言的安全机制来对 NF 进行内存的隔离（权限检查）和报文的隔离（不同 NF 不能同时处理同一个报，将报文传递给下个 NF 后，不再拥有对其控制权）。

此外 SafeBricks 还设计了一套可以用于远程交付 NF 代码和规则的机制。可以在 NF 的供应商不提供源码、NF 的使用者不泄漏规则，NF 运营商（云）不能获得 NF 的信息的情况下，进行 NF 的部署。首先有安全机制，NF 供应商和用户会提供加密的源码和数据，再有云端 SafeBricks 提供的编译器进行解码和编译。

这样的方法也有局限性：

1. 将多个 NF 放在同一个 Enclave 里虽然保证了整体处理性能的提升，但由于其 run-to-completion 的模型，无法保证各个 NF 之间的性能隔离，一旦 NFchain 上有个 NF 的开销增大，则整个 NF 链的效率由于瓶颈 NF 受到影响；
2. 继承 NetBricks 核心的 SafeBricks 会有先天的问题，即将一个 NF 拆成多个 NF instances 而放置在不同的 Enclave 里，会导致 NF 状态的共享问题（读写操作）

Metron

Metron 是 NSDI'18 的文章，主要目的是为了消除 NFV 场景下 NF 服务链的跨核的通信开销。将一部分的包的处理交给网络。主要方法是：

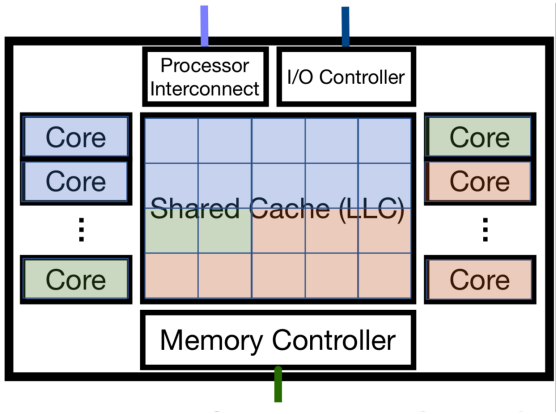
1. 在可编程硬件的交换机上安装 OpenFlow 和 P4 的规则，可以执行一些读写操作（与状态无关的，与状态有关的读写操作只能在 CPU 上完成）
2. 先用 SNF 的方法，对已知 NF 链中存在的多个同等类（每一类流都由相同 NF 链处理逻辑来处理，可以将这些 NF 合并成一个大的 NF，并放置在同一个核上。这样可以减少核之间的交换开销）使用交换机提前对流量进行分类/标记；
3. 这样将流分类的方法也可能导致流量的不均衡。针对这种情况，Metron 会根据流量的统计情况将较大的等价类拆分成多个等价类，再由多个核来分别执行

已有的方法中，有使用专用的核来跑软件交换机，以达到区分和分配流的目的；还有用 RSS 来分配流，但处理报文的 NF 链与接受报文的不在同一个核中，这样也会带来开销；另外还有 Intel Flow Director 这种硬件也可以实现类似 RSS 的功能；但是 RSS 不是很能均衡流的大小，可能让大流集中到某些核上去（基于 Hash 规则）；FD 的分类能力比较有限（match-action），一旦没有 match 项，则得专门处理。

我认为还有以下的问题：

1. 我们看到，这种将同一个 NF 逻辑复制到多个核上内存的方法，虽然能够让不同的流进入不同的核处理；但是我们知道 NF 的状态分为两类，一种是每流状态（每流计数器），还有一些全局的状态（防火墙规则）；如果对于每流状态，这种方法可以将不同流的状态分开，并放在不同的核中，是不需要锁的；但是对于全局状态，则这种方法会导致核之间竞争资源，争夺锁，也会让性能下降。对于将同一个等价类分成多个细小等价类，并放置在多个核处理时，也会导致上述问题，因此我认为 Metron 适合的是每流状态的 NF。
2. 对于 scaling 问题，如果流的变化较快，那么进行重新分配的代价也是比较大（重新安装规则，而且可能涉及到对状态的迁移，原来在同一个核上的状态被重新安排到其他核）

ResQ



ResQ 的题目是 Enabling SLOs in Network Function Virtualization。这里 SLO 是 service level object。针对此文想要实现的目标是可保障/可确定的性能（performance）和延时（latency）。缓存一般分为三级，L1 最小最快，为每个核独享，L2 为几个核共享一个，L3 则为全局共享。其认为运行在不同的 core 上的 NF 之间在 LLC 上存在竞争，提出通过对缓存进行合理隔离是可以做到保障的。ResQ 为 NFV 合理高效使用 Intel Cache Allocation Techonology（CAT）技术提供了优化方法。CAT 可以将 Cache 合理分配到每个，以保证隔离。但新的硬件方案 Intel Data Direct I/O (DDIO)能够将以太网帧直接从网卡上通过 DMA 传输到 cache 上。这样不同的核会共同同一块 IO buffer，会造成竞争，而无法实现部分 SLOs。因此 ResQ 提出通过控制 NIC 和 cache 之间的 DMA 的 buffer（DPDK 和 netmap 均实现）来保证不同的 core 之间占用 LLC 的隔离性。ResQ 设计了调度器，为 NF 提供静态需求和弹性需求。目前没有看到文章有什么短板，因为文章比较 tricky，能找到 LLC 的问题，是一种 NFV 场景下，对 packet IO 的优化。