Linux 内核的 Go 语言实现研究

柴艳娜
(长安大学 信息与网络管理处,陕西 西安 710064)

摘 要:现代操作系统都会通过内核管理和调度各种硬件资源,内核的安全稳定是操作系统的基础。现代成熟的操作系统内核均由诸如 C 等低级语言进行开发,比如 Linux 内核。虽然 C 语言的底层操作能力对内核很有帮助,但是也因此带来许多缺陷和漏洞。用高级语言开发的内核可以规避很多潜在风险,但是可能付出的代价是性能的降低。完整地实现一个内核是一项巨大的工程,因此该文用 Go 语言实现内核的网络堆栈子系统,以此研究用高级语言实现操作系统内核的可行性,评估用高级语言开发内核的优势和缺点。该文主要实现的网络协议包括 Ethernet,ARP,IPv4,ICMP,UDP 和 TCP,并且会从代码可读性和性能两个角度与 C 语言开发的网络堆栈子系统进行比较,从而分析 Go 语言开发操作系统内核的可行性。

关键词:Linux 操作系统;内核;网络堆栈;Go 语言;TCP/IP 协议栈
中图分类号:TP31 **文献标识码**:A **文章编号**:1673-629X(2021)06-0070-06
doi:10.3969/j.issn.1673-629X.2021.06.013

Research on Implementing Linux Kernel with Go

CHAI Yan-na
(Dept. of Information and Network Management, Chang'an University, Xi'an 710064, China)

Abstract: Modern operating systems manage and schedule various hardware resources with Kernel, so it's the base that the built-in kernel works well and securely. The modern mature kernels are developed by lower-level language such as C, like Linux kernel. Although the low-level functionalities of C language are helpful to the kernels, they also give rise to many classes of bugs. Kernels written in higher level languages avoid many of these potential issues, at the possible cost of decreased performance. This research evaluates the advantages and disadvantages of a kernel written in a higher-level language. To do this, the network stack subsystem of the kernel was implemented in Go and compared to a representative network stack written in C. Modules for the major networking protocols, including Ethernet, ARP, IPv4, ICMP, UDP, and TCP, were implemented. The analysis of code readability and performance on two network stacks would be made and the conclusion of Go is a viable alternative to C in kernel development would be demonstrated.

Key words: Linux; kernel; network stack; Go programming language; TCP/IP suite

1 概 述

计算机是现代日常生活的一种必需品,其高效可靠的运行需要依赖于一套稳健无缺陷(bug-free)的操作系统。现代操作系统都会使用内核(kernel)来对硬件进行管理,因此可以说内核的安全稳定决定了人们与计算机相处的体验。内核中的缺陷(bug)将可能使用户的应用程序甚至操作系统本身变得不可靠[1]。

内核是用户和应用程序与计算机硬件之间的桥梁,内核管理各种系统资源,包括内存和硬盘空间,并且处理 CPU 处理程序的调度。它也提供对输入输出设备和网络的访问。应用程序运行在内核之上,通过内核的系统调用从而使用到内核的功能。

1.1 现代内核的问题

大多数成熟的操作系统内核都是用 C 语言实现的。C 语言因为允许高度控制内存使用以及其他如可与汇编语言互操作等低级程序操作特性,成为最受欢迎的内核语言[2]。这种高度的自由也会付出一些代价,比如内存释放两遍的错误、数组越界的错误以及死锁[3]。同时它也不能防止数据类型的错误解析,保证不了类型的安全性。

随着计算机多处理器以及多核处理器的增加,如何高效地利用多线程是评价内核优秀与否的一个有力因素。C 语言实现的内核不能轻易地全面发挥多核的性能,因为 C 语言本身没有涵盖现代处理器的特性,C

收稿日期:2020-07-23 **修回日期**:2020-11-24
基金项目:陕西省信息化重点建设项目(2171-20120042);陕西省网络态势感知平台建设项目(0031-214024160305)
作者简介:柴艳娜(1984-),女,硕士,工程师,研究方向为计算机网络应用与信息技术、网络管理与安全等。

中的线程(thread)对内存和 CPU 来说都是很昂贵的一笔开销,而线程之间的同步机制则更复杂,所以内核需要大费周章地实现一套机制来充分调动多核计算机的全部性能[4]。

如果用 Java,Go 等高级语言来开发内核,则可能会规避掉很多上述问题,比如许多高级语言提供了数组越界检查和内存垃圾回收机制。然而,通常来说高级语言开发的程序会比 C 语言的慢,有时候由于代码解释、自动内存管理、垃圾回收等特性,会带来很大的系统开销。同时,高级语言很难操作汇编语言,因此可能很难满足内核的底层任务调用。

1.2　现有社区项目

当下社区中有很多用高级语言实现内核的尝试,诸多原因导致了它们没有一个被广泛采用。

Mirage 是一个 Linux 基金会项目,致力于将 Web 应用变成一个运行在 Xen 虚拟机下的独立的专属精简内核(Unikernel),它包含一个用 OCaml 开发的内核子系统的早期实现。因为它是为专属内核(单用户单进程,大多运行于虚拟机中)开发的,所以不能满足大多数普通用户的需求。另外,也不能在多核上并行,因为它本来就是为单进程运行而设计的。

Pycorn 是一个用 Python 开发的操作系统,目前只兼容 16 位 ARM 微处理器。因为 Python 是一门解释型语言,Pycorn 实际运行十分慢,性能不是该项目的目标。因此它从未被广泛使用过。

1.3　内核子系统

因为实现一个完整的内核是一项巨大的工程,所以该文代之以实现一个内核子系统,即网络堆栈子系统,来进行相应的研究工作。网络堆栈(network stack)是任何内核必须有的特性,网络堆栈的功能和性能可以容易地比较和测试,因此是个比较理想的可用于研究的子系统。

1.4　Go 语言

该文用 Go 语言实现内核子系统,用于研究用高级语言开发内核的相对优势。之所以选择 Go 是因为语言本身自带的优秀的 CSP 并发模型(concurrent sequential processes)[4-6]。CSP 模型将复杂任务解构成更小的、更加可管理的子任务。这些子任务都能被单个进程所处理,进程之间彼此保持通信,共同完成原始的复杂任务。

CSP 模型的目标是帮助程序员设计,实现和验证复杂的计算机系统,这是十分重要的,特别是要设计一个如内核般复杂的软件。Go 提供了线程安全(thread-safe)方式的 CSP 模型,Go 语言的线程即协程(go-routines),同步的通信构造即通道(channel)[7]。Go 语言运行时自动根据计算机的物理内核数量来管理调度协程。CSP 模型能让人很容易地使用计算机所有内核,同时改善代码的可读性,使得更简单地进行调试和减少产生的缺陷。网络堆栈很自然地可以被划分成多个子任务去运行,可以充分利用 Go 协程去动态调度高效利用所有可用物理内核[8]。

CSP 模型只在垃圾回收语言里有可行性,Go 提供了必要的垃圾回收。Go 是一门强类型语言,能减少一大类错误,包括错误类型转换,内存释放两遍,对象释放后再使用等。Go 的延迟声明(defer statement)允许在函数结束时更方便地清理,减少那些疏于管理的资源导致死锁的可能性。

1.5　研究目标

Go 和 CSP 模型的优势可能伴随着某种代价,比如垃圾回收有性能花销并导致运行时的短暂暂停。另外,多核的使用,将带来昂贵的内核间通信。该文的目标是评估 Go 带来的收益是否能盖过性能损失带来的劣势。

2　实　现

该文实现的独立网络堆栈(项目代号 NStack)是建立在 Tap 虚拟网卡的基础上。为了功能完整,所有基础网络协议,包括以太网(Ethernet),ARP,IPv4,ICMP,UDP 和 TCP,都被实现。为确保性能不受影响,延迟(latency)和吞吐量(through-out)会被测试,并与 C 语言实现的网络堆栈进行比较。

2.1　Tap 接口

Tap 接口即一种虚拟网络接口(虚拟网卡),用软件来模仿实际硬件。NStack 会将 Tap 接口当作正常物理接口一样读写[9]。Tap 接口会关联一桥接接口,就好像一个路由器作为主机的一个子网接入其中,这样可以允许 NStack 能使用它自己的 MAC 地址和 IP 地址,连接到外部网络。

2.2　协议实现

NStack 会实现数据链路层,网络层和传输层的协议,每一层独立运行自己的协议,如图 1 所示。分层模型可以增加并行,在高负载下提供高效服务[10]。

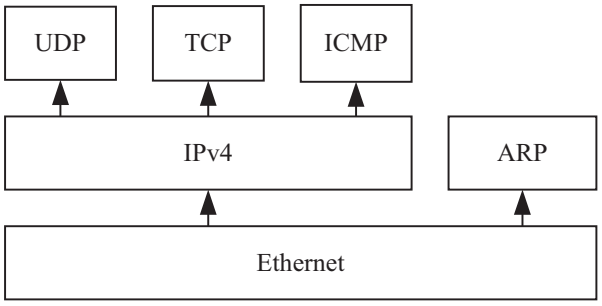

图 1　网络协议栈

每一个协议的实现使用了类似的结构,包处理器

(packet dealer)。IP 包处理器如图 2 所示。包处理器从低层级读取数据包,并通过通道传输。通道以箭头表示在图 2 中。IP 包处理器将数据包发给不同的 IP reader 协程。IP reader 处理完接收到的数据包后,将处理结果转发给下一层的包处理器。

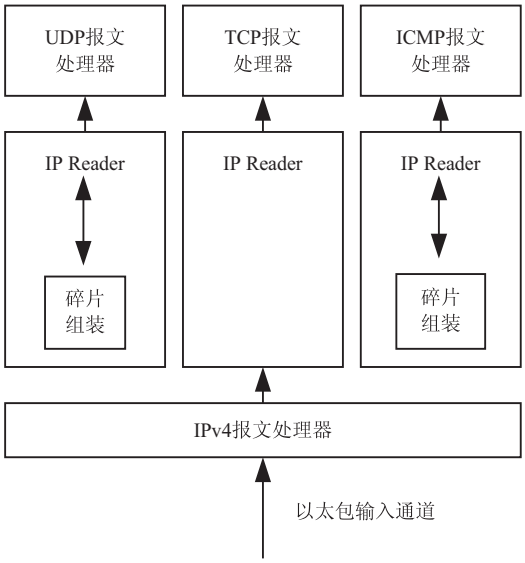

图 2 IPv4 包处理器

(1)以太协议层允许其他不同层的协议绑定到特定的以太协议。比如 IPv4 实现会绑定到以太协议 2048 去接收所有 IPv4 数据包,ARP 实现则绑定到以太协议 2054。

(2)地址解析协议(address resolution protocol, ARP)会被实现用于 MAC 地址的获取,数据的网络传输需要物理信息的支持。ARP 能让 NStack 从目标主机的目标协议地址中获取 MAC 地址。NStack 为每个 ARP 请求创建一个协程负责处理。处理时协程会被阻塞直到主 ARP 包处理器通知其响应或者请求超时。

(3)IPv4 的设计如图 2 所示,它使用包处理器结构,包含多个 IP 读取器和分片重组器。所有组件之间的通信都是通过通道进行,如箭头所示。

当 IP 包大小超过最大传输单元(maximum transmission unit,MTU)时,便会出现 IP 分片,IP 包会被拆分成多个分片,每一个分片都包含一些信息用以重组。当分片数据包到达目的主机,它们便会被重组成原始 IP 包。

NStack 的分片重组器演示了 CSP 模型的优点。每个分片重组器都囊括了对分片 IP 数据包的处理过程以及相应的数据。与用全局数据结构来管理所有分片数据包重组的传统方法相比,为每个报文分片分配一个专属重组器,这种 CSP 模型的做法可以大幅降低代码的复杂度。轻量级的 Go 协程设计让数据隔离变得可行,垃圾回收又大幅降低内存泄漏的可能。

(4)NStack 实现了 ping 及 ICMP 协议。ICMP 实现也是遵循包处理器结构。ping 实现也有其相应的包处理器,ping 的 ICMP 包会被 ICMP 包处理器先行处理,然后再发给 ping 的包处理器处理。ping 包处理器会将 ping 请求转发给一组特别的协程,用于回复 ping 请求。如果 NStack 已经发送了 ping 请求,则 ping 包处理器将会把回应转发给对应请求的专属协程负责。

(5)用户报文协议(user datagram protocol,UDP)是个无连接的协议,因为它相对简单,NStack 便用一个基础的包处理器将其转发给对应的 UDP 读取器。

(6)传输控制协议(transmission control protocol, TCP)是面向连接的传输层协议,它保证了数据传输的有序。因为 TCP 是面向连接的,所以它会需要服务端和客户端来初始化连接。一旦连接建立成功,便由传输控制单元(transmission control block, TCB)进行管理。

NStack 里 TCP 也是使用标准的包处理器结构管理源端口和目的端口,每个 TCB 里都有 2 个长期运行的协程。一个处理接收到的数据包,另一个则等待和发送数据,也会负责创建额外的协程管理数据包的重发,这 2 个协程便代表着半双工 TCP 连接。TCB 内部也会用到通道来同步和管理所有创建的协程。比如,处理接收数据包的协程发现收到一个确认数据包时,便会用通道通知数据包重传协程。

2.3 测 试

NStack 会与 Tapip 进行性能比较。Tapip 是一个由 C 语言开发的多线程网络堆栈。这个比较允许评估用高级语言开发网络堆栈的优点和缺点。两个网络堆栈都实现了相似的协议,都在用户空间(user space)操作,都使用 tap 虚拟接口。测试机器是 Ubuntu 14.04/Linux 3.13.0,16 GB 内存,Intel Xeon Quad Core Dual Socket 处理器。

2.3.1 延 迟

为测试延迟,将取 50 次 ping 响应时间的平均值作比较。测试环境的一台 Linux 虚拟机将运行两个网络堆栈,ping 请求从该虚拟机发出。为判断堆栈在负载增加情况下的性能,多个 ping 会被同时并发发送。从 1 个增加到 1 000 个并发 ping“连接”来模拟网络堆栈可能接受的负载。为保证对两个网络堆栈公平,其他的变量都将保持不变,包括每个 ping“连接”发送的 ping 请求数,ICMP 接受缓冲区大小以及 ping 请求数据包大小。

2.3.2 吞吐量

第二个将要评估的性能指标便是吞吐量。一个堆栈的吞吐量是在给定时间内,能发送或接收的数据量大小[11]。以下步骤将用以测量两个堆栈的吞吐量:

(1)初始化一个 TCP 服务端。

(2)初始化一个 TCP 客户端,连接会在 local 网络(localhost)中建立,以排除 tap 虚拟网卡导致的开销。

(3)客户端发送 4 KB 数据给服务端。

(4)计算堆栈完成上述过程的总时间,该时间和发送的数据量将用来计算吞吐量。

为测量堆栈的相对扩展能力,将会逐步增加客户端数来测量性能[12]。最大测试到 100 个并发客户端。有许多预防措施将用于保证吞吐量的准确测量,比如所有可比较的缓冲区大小都一致[13]。在 Tapip 中,每个客户端和服务端连接都运行在各自线程里,NStack 类似,但是用的是 Go 的协程而不是线程。另外,也会确保所有连接完成且连接的负载被完整传输之后再停止运行网络堆栈[14-15]。

3　结果分析

NStack 的代码与 Tapip 比较类似,但是从结果来看,性能上,包括延迟和吞吐量,NStack 相比之下出色得多。

3.1　准确性

NStack 和 Tapip 都能准确地运行协议,这可以通过分别测试两个协议栈与一台 Linux 终端的连接来进行判断。测试中发现 Tapip 有内存泄漏的情况。这是因为 Tapip 会开辟缓存区存储数据包,在某些情况下这些缓存区不会被释放或者重复释放。当缓存区被重复释放时,Tapip 会崩溃或者导致异常行为。当缓存区不会被释放时,Tapip 会不断侵占内存,直至系统崩溃。Go 则由于有内置的垃圾回收,可以很好地避免这种情况的发生。

3.2　代码比较

虽然很难量化地评估编写 Go 语言相比较 C 语言的优点,但是从一些代码片段的比较还是可以看出高级语言的某些优势。以下以 IP 报文分片重组的处理代码举例说明。

(1)当新的 IP 分片到达时,需要初始化分片重组器。Tapip 则会使用全局结构体存储所有待重组的数据,C 代码如下所示:

```
struct fragment *frag;
frag=xmalloc(sizeof(*frag));
list_add(&frag->frag_list, &frag_head);
list_init(&frag->frag_pkb);
return frag;
```

NStack 会给每个待重组的包新建一个 Go 协程,Go 语言代码如下:

```
ipr.fragBuf[bufID]=make(chan []byte, FRAG_ASSEM_BUF_SZ)
quit:=make(chan bool, 1)
done:=make(chan bool, 1)
didQuit:=make(chan bool, 1)
go ipr.fragAssembler(/* ... */)
go ipr.killFragAssembler(/* ... */)
```

(2)当添加分片到重组队列时,Tapip 的 C 语言代码如下:

```
int insert_frag(/* ... */){
/*一些额外的分片处理 */
list_add(&pkb->pk_list, pos);
return 0;
frag_drop: free_pkb(pkb); return -1;
}
```

Go 语言代码则如下:

```
ipr.fragBuf[bufID] <- b
```

Go 可以用协程处理 IP 报文分片,因此它可以简单地将分片转发给对应的协程处理,同时可以紧接着处理后续数据包。此举会改进 NStack 代码的模块性、可读性和并发性。

(3)分片处理完成时的 C 语言代码片段如下:

```
if (complete_frag(frag))
pkb=reass_frag(frag);
else pkb=NULL;
return pkb;

struct pkbuf *reass_frag(
struct fragment *frag){
/* more processing */
delete_frag(frag);
return pkb;
}
```

Go 语言代码片段如下:

```
ipr.incomingPackets <- append(
fullPacketHdr,payload...)
done <- true
```

经过对比,可以凸显出 Go 语言以及 CSP 模型的优势。Tapip 必需按顺序处理数据包,在前一个数据包未处理完时,下一个数据包只能在缓冲区中等待。这会带来一些问题,比如这便需要 C 语言的 IP 实现去跟踪所有正在进行的分片重组的状态,这样不可避免地会使用全局变量和结构体来记录共享信息,并且会让线程同步变得困难。NStack 与之相反,它会对接收到的每个分片 IP 包创建一个独立的分片重组器协程,每个协程各自负责独立的分片组装成 IP 片段。分片重组器处理重组完数据包后,它便简单地将重组片段发回后续的 IP 数据包处理过程。IP 数据包这个主处理过程与分片重组器是独立的协程,因此可以实现完全的并行和并发,代码也更简洁可读。

(4)在清理分片时,C 语言的 Tapip 需要显性地释放每一个内存缓存区,代码如下:

```
struct pkbuf * pkb;
list_del(&frag->frag_list);
while (! list_empty(&frag->frag_pkb)) {
pkb = frag_head_pkb(frag);
list_del(&pkb->pk_list);
free_pkb(pkb);
}
free(frag);
```

而 Go 语言只需跟踪通道即可:

```
delete(ipr.fragBuf, bufID)
```

Go 语言的简洁友好可读由此可见一斑。

3.3 延 迟

1 个 ping 请求时,Tapip 的 0.074 ms 优于 NStack 的 0.234 ms,但是随着并发请求的增加,当 1 000 个 ping 请求时,NStack 的延迟为 0.717 ms,差不多比 Tapip 的 3.279 ms 好 5 倍。NStack 在连接数为 600 时,开始领先于 Tapip。NStack 延迟的增加是线性的,而 Tapip 是指数型的。NStack 的延迟趋势是优于 Tapip 的,因为在请求数很少时,两者之间延迟的差距很小,可以忽略不计,但是在大量并发 ping 时,差异就明显变大,如图 3 所示。

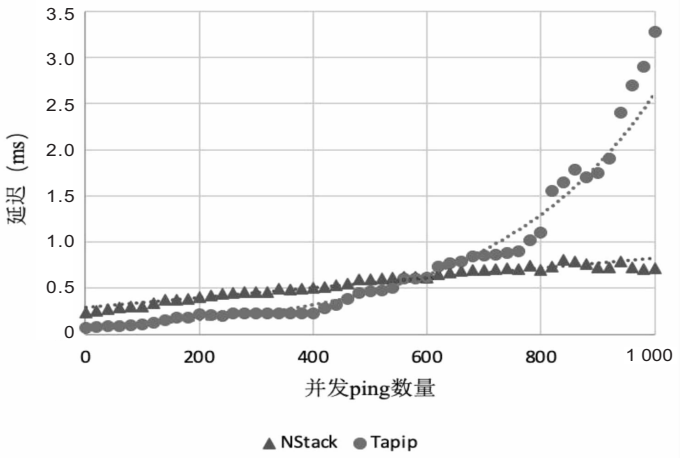

图 3 并发延迟性测试结果

基于图 3 的结果,可以得出 Tapip 能非常快地处理小量级的数据包,而对于大量的数据包涌入时,则显得处理乏力,性能极差。相对应的,NStack 会用相对较长的时间来处理每个数据包,但是因为其在每个协议实现中良好的并发控制,在负载大量增加的情况下,几乎不影响其处理性能。表现出来便是结果中,Tapip 虽然开始性能优秀,但延迟却随着并发量的增长,迅速增大上升,而 NStack 则小幅平缓的增加。Tapip 陡峭的增长趋势凸显了其底层架构的问题,即在所有的协议层处理完一个数据包后,再处理下一个数据包,这种做法不是一个高效的方法,因为这会导致扩展或并发难以实现。

3.4 吞吐量

测试结果如图 4 所示,1 个并发连接时,NStack 的吞吐量达到 7.3 Mbit/s,而 Tapip 的只有 4.6 Mbit/s。当 100 个并发连接时,NStack 达到了 284.9 Mbit/s,而 Tapip 则只有 195 Mbit/s。并且,NStack 的吞吐量增加速度比 Tapip 快得多。这表明 NStack 可以继续在更大量的并发情况下扩展吞吐量而 Tapip 则很可能处理不了这种负载。

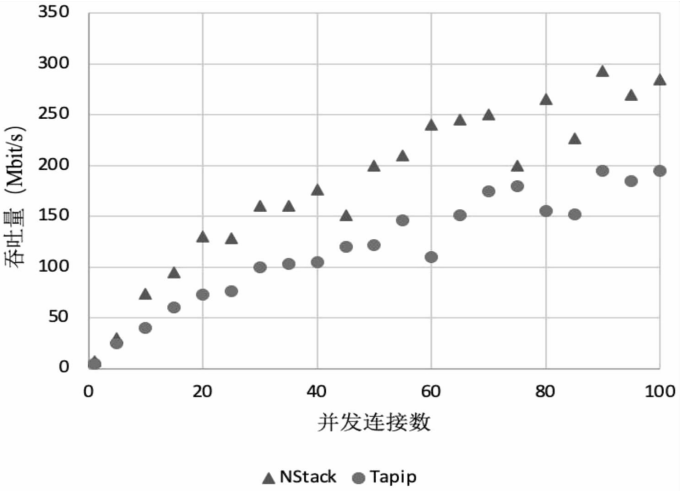

图 4 并发吞吐量测试结果

结果有力地验证了 NStack 的架构。在 Tapip 里,所有的传输控制块(transmission control block,TCB)都是由单个线程管理的;相应的,在 NStack 中,每个 TCB 由两个线程进行管理,分别负责一半的上下行连接,因而 NStack 可以更高效地在有限的 CPU 核数上多路复用大量的连接,可以达到更大的吞吐量。在小量并发连接时,NStack 也工作地更高效,因为它把 TCB 的处理工作拆分为两个 Go 协程,而 Tapip 则自始至终都是一个线程执行处理任务。

4　结束语

操作系统内核对于管理计算机系统资源而言是十分重要的核心组件,如何在兼顾性能的前提下,引入高级语言进行开发,降低低级语言开发内核带来的复杂性和安全隐患是该文的初衷。该文以内核的网络堆栈子系统为出发点,用 Go 语言实现 NStack,研究高级语言开发内核的可行性和便利性。NStack 和对比实验的 C 语言开发的 Tapip 都是基于 tap 接口,都实现了相类似的协议,比如 IPv4,ARP,UDP 和 TCP。在延迟性和吞吐量的对比实验中,可以发现 NStack 有优秀的性能表现,在延迟性测试中,当并发数大于 600 时,NStack 取得更低的延时;在吞吐量的测试中,NStack 的并行化让其在所有的测试场景中都取得了优于 Tapip 的吞吐量。

实验表明,Go 语言带来的简洁和模块化可以提供优于 C 语言的帮助,用 Go 开发内核子系统可以改善代码的可读性和可靠性,结构模块清晰,良好的并发能力和稳定性,同时又对内核整体性能没有产生重大不良影响。结果表明,对于内核开发来说,Go 语言可以是一个重要的 C 语言替代者。

参考文献:

[1] CHODOREK A,CHODOREK R R. Light-weight congestion control for the DCCP: implementation in the Linux kernel [M]//Data-centric business and applications,towards software development (Volume 4). [s. l.]:[s. n.],2019:245-267.

[2] 汪微微,郝　田. 基于 Linux 内核实现 IP 层流量均衡的方法[J]. 计算机产品与流通,2018(6):43.

[3] 周　丹,陈楚康,蔡万强,等. 基于 Linux 内核的用户态网络协议栈的实现[J]. 信息通信,2019(7):200-204.

[4] 王晓峰. Golang 语言实现的流水线模型[J]. 电子技术与软件工程,2020(1):53-54.

[5] 刘艳平. Go 语言实现数据库驱动的方法[J]. 计算机与现代化,2018(1):113-115.

[6] 雨　痕. Go 语言学习笔记[M]. 北京:电子工业出版社,2016.

[7] BOKOR Z L. The Go programming language[J]. IEEE Software,2016,31(5):104.

[8] 陈　麓,柏　雪. Linux 下基于 MII 接口网络驱动的设计[J]. 数字化用户,2018,24(46):17-18.

[9] CHAUHAN S,CUTHBERT D,DEVINE J,et al. Network performance[M]//AWS certified advanced networking official study guide. New York:Sybex,2018.

[10] SCHOEBERL M,PEZZAROSSA L,SPARSO J,et al. A multicore processor for time-critical applications[J]. IEEE Design & Test of Computers,2018,35(2):38-47.

[11] YU Z,ZUO Y,ZHAO Y. Convoider: a concurrency bug avoider based on transparent software transactional memory [J]. International Journal of Parallel Programming,2020,48(1):32-60.

[12] SCHOEBERL M,PEDERSEN R U. tpIP: a time-predictable TCP/IP stack for cyber-physical systems[C]//2018 IEEE 21st international symposium on real-time distributed computing (ISORC). NTU,Singapore:IEEE,2018.

[13] 胡守峰,陈文俊,黄光明. 基于轻型协议栈 LwIP 的 LXI 接口实现[J]. 电子测量技术,2019,42(20):114-119.

[14] 严博文. 设计与实现一种面向协议栈特征的测试程序[J]. 兰州理工大学学报,2019,45(3):108-112.

[15] 徐　健,孙　庆. LwIP 协议栈的 pbuf 结构探索与研究[J]. 单片机与嵌入式系统应用,2018,18(2):14-17.