多媒体流的轻量级 TCP/IP 协议栈的优化①

陈 慕, 曾培峰

(东华大学 计算机科学与技术学院, 上海 201620)

摘 要: 提出了一种在轻量级 TCP 协议栈上实现的零存储丢包重传机制, 并设计了一种专门适用于多媒体视频流的坚持计时器. 这两种机制的共同作用可以有效提高多媒体数据流在嵌入式系统上传输的效率和稳定性, 避免网络拥塞. 同时, 针对不同的传输环境,对改进后的 TCP/IP 轻量级协议栈进行了效率和稳定性的测试, 测试结果表明, 经优化后的协议栈在不同的网络拥塞可以保持正常工作.

关键词: 丢包重传; 坚持计时器; 多媒体视频流; 嵌入式系统; 轻量级协议栈

Optimization Technology for Light Weight TCP/IP Stack Based on Multimedia Streaming

CHEN Mu, ZENG Pei-Feng

(College of Computer Science and Technology, Donghua University, Shanghai 201620, China)

Abstract: This paper proposes a zero-copy packet loss recovery mechanism on light weight TCP/IP stack. Meanwhile, a new Persistent Timer especially for multimedia streaming is also described. With these two mechanisms, the multimedia streaming transmission could work more effectively and stably on embedded system, avoiding the network congestion. Besides that, an efficiency and stability test is proposed for this improved light weight TCP stack under various transmission situations. Experimental results show that the optimizated light weight TCP/IP stack could keep working under different network congestion.

Key words: packet loss recovery; persistent timer; multimedia streaming; embedded system; light weight protocol stack

网络的带宽正在以指数型倍数不断提速, 这种提速带动了多媒体服务与应用的惊人的发展. 与此同时, 嵌入式系统在 Internet 领域中的飞速发展也正迎合了用户对高速网络和便捷性平台需求的不断提升, 近几年的很多研究都致力于嵌入式系统上多媒体流传输和播放上的应用. 然而作为应用最广泛的传输协议 TCP 协议, 却并不被看好适用于多媒体流传输. 多媒体流传输以它实时性的特点才深受人们喜爱, 而 TCP 协议在传输时引起的端到端的延迟和抖动会严重影响流媒体的连续性. 正是因为这种限制, 以往的很多研究都倾向于基于 UDP 流媒体传输协议的算法改进和协议扩充, 例如, RTP, RTCP[1-2], TFRC[3]. 然而, 虽然传统研究表明 TCP 不适合于流媒体传输[4], TCP 由于其自身无法代替的优点, 仍然是商业流媒体传输中应用范围最广的协议. Sripanidkulchai 等人在 2004 年研究了一个大容量的分发网络的实时流媒体的工作负载, 指出大概 40%到 80%的实时流媒体使用的是 HTTP/TCP 协议[5]. 此外, Van der Merwe 等人在 2002 年分析了两个商业流媒体服务器在 4 个月的时间段内 450 万对话的日志并发现 72%的请求流媒体和 75%的实时流媒体的传输都使用了 TCP 协议[6]. 与此同时, Bing Wang, Jim Kurose 等人通过数学模型和具体实验, 确认了在某些情况下(当 TCP 吞吐量大约是多媒体比特率两倍的时候), TCP 可以提供足够好的媒体流传输效果[7].

近年来微电子技术的不断进步让越来越多的功能被集成到单个芯片上, 引发了人们对嵌入式系统的更进一步的需求. 嵌入式系统具有的尽可能小的内存消耗和尽可能小的代码量的特点, 十分符合计算机技术更简单更迅速的发展方向. 但由于硬件资源的限制, 人们一直无法将一个 PC 端完整的传输系统结构实现

① 收稿时间:2014-07-30;收到修改稿时间:2014-09-09

在嵌入式系统上. 因此, 很多研究[8-10]选择在嵌入式系统上移植或设计轻量型协议栈, 但很多轻量级协议栈只能实现一些最基本的传输功能. 对于嵌入式系统来说, 无论是基于 UDP 协议栈的扩展, 比如 RTP, TFRC, 还是在客户端加强对流媒体数据的处理技术[4], 都不能起到很好的作用, 并且很可能会影响到嵌入式的零拷贝机制. 零拷贝是让传输数据只在用户缓冲区和 NIC[11]之间交换的一种机制, 它需要例如 DMA 控制器这种硬件支持设备. 由于在传输过程中没有发生数据拷贝, 零拷贝机制可以减少内存和 CPU 处理的消耗, 让更多的资源服务于传输和计算. 综合上述的各种问题, 基于嵌入式系统的多媒体流传输的改进工作应该致力于轻量级 TCP 协议栈的功能扩展和完善.

很多研究[8-12]用多种方法将传输协议实现于嵌入式系统上. 文章[8]选择将 Linux OS 作为整个的嵌入式系统. 然而, 移植一个未裁剪的 Linux OS 会对 cortex ARM 芯片造成很大负担, 系统需要消耗大量的硬件资源来运行它. 所以多数人使用 lwIP 的 tcp 协议栈移植[8], 为了方便运行, 这些轻量级的协议栈通常将拥塞控制, 丢包重传以及坚持计时器等机制裁减掉, 但这些裁剪后的协议只能实现一些最基本的传输功能. 我们知道, 当短时间内大量数据流涌入到网络时会引发严重的丢包情况, 应用于 PC 机上的传统的协议栈使用很多缓冲区来保存发送过的包, 实现丢包重传和拥塞控制机制. 而上述那些轻量级协议栈由于没有拥塞控制和丢包重传机制则会引发网络连接崩溃[13], 它们只能通过重置连接或限制带宽来应对这些问题. 除此之外, 在没有坚持计时器的情况下, 当更新窗口 ACK 值的包丢失后, 网络连接的两端会进入等待数据包的死锁状态, 网络连接会因为超时而终止[14]. 这些问题会严重影响传输的效率和稳定性, 上述的轻量级协议栈由于裁剪掉了很多功能, 在传输时不可避免地会碰到这些问题.

1 系统概述

本文在保证零拷贝的前提下对于轻量级 TCP 协议栈进行了优化, 让轻量级 TCP 协议栈在嵌入式系统上传输数据时可以实现丢包重传和拥塞控制等操作. 不仅解决了由于数据包丢失引起的播放质量和效率的降低, 同时保证了更多的 CPU 开销可以用于图像处理. 同时, 针对更新窗口 ack 包丢失问题, 本文针对多媒体流进行了特殊处理, 让协议栈在问题出现时仍可顺利传输多媒体数据流.

本文的其余部分安排如下: 第二章着重介绍一种应用于轻量级协议栈的基于零拷贝的丢包重传算法; 第三章着重介绍了一种专门适用于流媒体传输的坚持计时器; 第四章对本文中优化过的轻量级操作系统进行了多种情况下的效率和稳定性测试; 最后第五章总结全文.

2 丢包重传

2.1 零拷贝传输

零拷贝是指在数据包传输过程中, 内存不进行任何数据的拷贝. 在我们的 cortex ARM 单片机系统中, 我们有四个 descriptor 来处理数据. 四个 descriptor 组成一个环形结构.

传统的协议栈通常将数据从 descriptor 中拷贝到内部 Flash 内存中, 这是不符合零存储理念的. 在我们的系统中, 当媒体文件被发送时, 系统将 Tx DMA descriptor 直接指向数据所在位置, 而在接受文件时, 系统从 Rx DMA descriptor 缓冲区中直接获取包信息. 整个系统使用几个的物理内存空间和指针来实现零拷贝传输.

2.2 数据结构

首先简单介绍一下我们的多媒体文件结构. 当终端接受多媒体数据包的时候, 它需要判断每个包是属于哪一个待播放的媒体文件队列的. 为了合理划分, 我们的文件结构由 3 种多媒体数据包组成: 数据头包, 数据包和数据尾包. 对于数据头包来说, 媒体文件会在 TCP 头部和数据信息之间插入 4 字节的头文件标识和 4 字节的媒体文件长度. 数据包则是普通的包文件. 而数据尾包是每个媒体文件的最后一个包, 它通常不会是满包, 它的大小则由一个媒体文件在一个 MSS 中剩余的数据量来决定的. 接收端根据这种文件结构播放多媒体.

为了保证零存储模式, 我们开发了一自适应的循环传输列表来实现丢包重传, 它只占用很小的代码量和内存消耗. 经研究发现, 大部分的接收端缓冲区的空间大小为 65535 字节, 而我们的多媒体文件的平均大小是 11k, 考虑到 MSS, 多媒体头包以及尾包的大小, 我们将循环列表的长度定位 64, 这样可以用按位与运算来实现循环, 进一步减少计算量.

列表的每一项中含有 4 个参数: *media_length*(2 字节), *media_ptr*(4 字节), *Presend*(1 字节), *mediasize*(4 字

节). *media_length* 保存数据包中数据的长度, *media_ptr* 指向该数据包中数据在多媒体区域外存中存放的位置, *Presend* 用来记录当前数据包是否是多媒体头包, 如果是头包, *mediasize* 将保存该多媒体文件的大小. 同时, 我们还在 socket 中设定了 *pac_num* 和 *ack_pac_num* 两个参数来标记当前发送包和确认收到包的位置. 当一个包被发送时, 它的相关信息会保存到循环列表的第 *pac_num* 项中, *pac_num* 做循环递增, 每当收到一个可接受的 ack 包, *ack_pac_num* 便会指向它.

通过这种结构, 我们可以用较小的内存空间来实现其他设备整个接收端缓冲区的数据包存储效果, 节省了大约 90 倍的内存空间.

2.3 丢包重传和拥塞控制

通过上述的数据结构, 我们可以准确地将发送数据包的关键数据保存下来. 之后, 我们使用慢启动和快速重传机制来实现丢包重传和拥塞控制, 用参数 slowSent 和 slow 来分别描述已发送但未被 ack 的数据包的数量以及 CWND 的大小. 整个流程如下:

① 检查网络的拥塞情况, RWND(接收端通告窗口)的大小, 以及 DMA 状态来确认是否可以发送数据包. 检查系统状态, 如果当前是 FRetrans(快速重传)状态, 转入②, 如果是正常状态, 转入③;

② 关闭新文件的传输并重传丢失的数据包;

③ 正常发送新的多媒体文件.

当接收到新的 ACK 包时, 系统转入④;

④ 检查接收到的 ACK 包是否确认了之前发送过的数据包, 如果是, 转到⑤, 如果是一个 duplicate ACK 包, 系统将转入⑥

⑤ 计算接收端正确接受了多少个数据包, 将 *ack_pac_num* 指向之前提到的传输循环列表中相应的位置. 然后更新参数 *slow* 和 *slowSent* 的值, 以慢启动方式加快传输速度.

⑥ 计算收到的连续 duplicate ACK 包的数量, 并根据数量判断是否应该触发快速重传机制

如果系统进入步骤②, 说明传输过程中有一些数据包丢失, 网络处于拥塞状态. 在这种情况下, 系统将停止新的多媒体文件的传送, 然后根据上一个被正确确认的包的信息, 在传输队列中找出丢失的数据包. 系统会利用参数 *opposite_ack*(保存当前正确接受到的包的 ack 值, 极为当前需要重传包的序列号)以及媒体文件长度值 *mediasize*(如果是头包的话)以及其他的一些信息来重组需要重传数据包的头部, 同时利用 *media_ptr* 来指引重传数据包的数据的所在位置. 每次发送重传包后我们都会将 *opposite_ack* 递增包的长度来获取下一个重传包的序号值. 当所有丢失的包都被重传后, 系统将返回正常传输状态.

在步骤⑥中, 如果接收到超过 3 个(个数可以自己设置)连续相同的 duplicate ACK 包, 基本可以确认网络发生了丢包, 传输速度很可能超过了网络所能承受的最大强度. 这时我们需要降低传输速度并重传丢失的数据包. 系统会存储丢失数据包的数量, 初始化参数 *slow* 和 *slowSent* 并转入快速重传状态.

上述的整个过程中没有发生数据拷贝, 我们只用原本 1/90 的空间就可以让传输队列囊括 64 个数据包的所有信息. 系统在队列中标记了上一个发送的包和上一个被成功 ack 包的位置. 进入快速重传状态后, 系统从最后一个没有被成功 ack 的包开始重传, 用 *opposite_ack* 来作为重传包的序号, 让 descriptor 通过 *media_ptr* 从缓冲区地址中将相关包数据取出. 由于整个重传过程没有数据拷贝, 这种方法只用很小的 CPU 消耗就可以解决丢包重传问题, 将更多的资源用于多媒体采集等处理.

2.4 包乱序问题

即便是来自同一次连接的包在传输时也可能会由不同的路由器转发. 所以有时序号靠后的包反而会先到达, 这些包被称为乱序包. 接收端在接收乱序包时发现它们不是当前需要的包, 会回复 duplicate ACK 包. 如果这种情况连续发生, 则会引发网络的伪快速重传问题. 这种现象在有线网络中很少出现, 但在无线网络中发生较为频繁. 传统的有线 TCP 协议栈通常会忽略这种情况, 当做是正常丢包, 进行重传操作. 很明显, 这会牺牲很大一部分 CPU 和内存资源.

我们在协议栈中增加了一个机制来解决这个问题. 当一个可以被正常接收的 ACK 包到达时, 系统会检测当前状态. 如果已进入快速重传状态则说明发生了伪重传, 系统会停止重传并切回正常状态发送新的媒体文件. 这种机制可以尽快使系统恢复正常并减少不需要的重传工作.

3 坚持计时器

3.1 窗口探查

传统的窗口探查是发送一个字节的数据来让对方发送更新过窗口大小后的 ack 包. 但对于流媒体传输来说, 一个非图片数据的单字节信息可能会引发客户

端播放时的错误甚至崩溃. 而且过多的单字节图片会给播放端造成较大的困扰. 考虑到我们系统在不影响播放质量的情况下在拥塞控制中为接收端窗口预留了 1/4 的大小. 经过多次实验, 为了适应流媒体传输, 我们这里将窗口探查设定为下一个图片数据包.

3.2 指数型退避计时器

传统的时间退避算法是在在检测到窗口较小引发的传输暂停后的每 1.5*2n 秒(n 为发送创口探查的次数)发送一个窗口探查. 我们在实际操作中发现, 较长的等待时间会造成图像播放的停顿甚至崩溃. 显然, 我们需要一种更适合流媒体的计时器. 通过实际测量, 同时也为了不影响播放的质量, 我们设定 1.5 秒为一个探查窗口的发送间隔. 变量 *Persistent_Timer* 用来记录坚持定时器启动每次启动时的时间, 变量 *Persistent_Timer_count* 用来记录窗口探查的发送次数, 变量 *SysTime* 用来记录系统当前时间.

3.3 窗口更新 ACK 包的丢失

在处理窗口更新 ack 包丢失的问题时, 我们的解决流程如下.

① 当系统检测到对方的发送端窗口小于 1/4 时进入 step2, 否则进入 step5

② 检查 *Persistent_Timer* 的值是否为 0, 如果是, 进入第 3 步, 否则, 进入第 4 部.

③ 将 *SysTime* 的值赋给 *Persistent_Timer*, 退出图片发送过程.

④ 系统判断当前情况应该发送一个窗口探测还是重置连接.

⑤ 初始化变量 *Persistent_Timer* 和 *Persistent_Timer_count* 的值. 发送图片

在第④步中, 系统会检查当前已经发送的窗口探测的数量. 如果多于 9 个窗口探测已被发送, 说明这次问题不是由于丢失更新窗口 ACK 包引起的, 系统会通过 UDP 协议给接收端发送一个重置连接的请求. 如果不是, 系统会比对 SysTime 和 *Persistent_Timer* 的差值, 若大于 1.5 秒, 将 *SysTime* 的值赋给 *Persistent_Timer*, *Persistent_Timer_count* 加 1, 发送一个图片数据包, 若小于 1.5 秒, 退出图片发送过程.

4 测试

4.1 测试环境

在整个测试过程中, 本文所述的轻量级协议栈作为多媒体流文件的发送端. 这套嵌入式系统的芯片使用了 cortex STM32F407, 镜头的型号是 OV5642. 接收端是一台装有 32 位 Win7 操作系统的 PC 机, 使用我们自己编写的程序播放从单片机上抓取的多媒体文件. 接收端和发送端通过几个路由器连接在一个局域网内, PC 机的网卡型号是 Intel 82578DM.

4.2 测试过程和结果

图 1 是传统轻量级 TCP/IP 协议栈和我们经过丢包重传和坚持计时器处理的协议栈在多媒体传输播放系统里网络吞吐量的对比. 从结果中我们可以获悉, 经过优化后, 多媒体数据的传输并没有因为新增拥塞控制盒丢包重传产生的计算量而降低效率. 同时, 由于启用了慢启动拥塞控制, 在第 2 到 4 秒这个时间段, 优化过的协议栈可以更早地到达吞吐量的峰值, 使传输更为高效, 而且效果也十分显著.

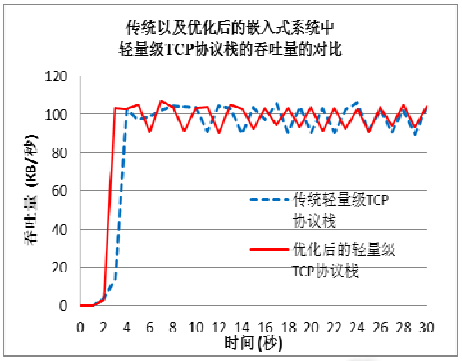

图 1　传输效率测试结果

图 2 是我们优化后的协议栈在不同丢包情况下的表现的结果. 我们在第二种情景中(蓝线)每隔 128 个包丢掉一个 DMA descriptor 传送的图片数据包, 在第三种情景(绿线)中, 在每 128 个包中丢弃连续的 10 个图片数据包. 从结果我们可以看出, 情景 2 的丢包对我们优化后的协议栈几乎造不成影响, 协议栈对丢包进行重传后, 播放过程仍可以继续进行, 只是在每次丢包的时候吞吐量的最低值从 90 降到了 80. 而在情景 3 中我们可以看到, 经过我们的优化后, 频繁的大面积丢包并不会造成连接中断和播放崩溃, 即便是在大量的重传工作后, 流媒体传输可以在很短的时间内由重传时的值(50)恢复到最大值. 与之相反, 从传统轻量级协议栈的测试结果来看, 普通的丢包就可能会造成连接和播放的中断和死锁, 系统需要等待指定时间来 reset 连接. 我们可以看到我们的协议栈表现优秀.

为了测试坚持计时器, 我们在窗口较小时对 pc 端

回复的 ack 包进行了丢弃并实时监测协议栈动态. 从测试结果来看, 坚持计时器会每隔 1.5 秒发送一个窗口探查, 接收端在接收到窗口探查后会继续发送更新过窗口的 ack 包, 这有效解决了 ack 包的丢失问题.

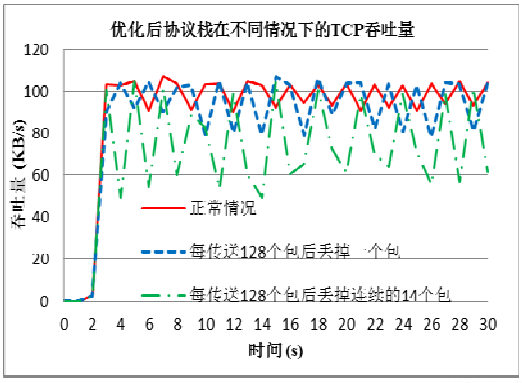

图 2 协议栈在不同丢包情形下的表现

3 结语

本文提出了一种新的轻量级 TCP 协议栈, 它不仅可以处理丢包重传问题, 并拥有适用于多媒体传输的坚持计时器. 本文主要包括了以下几个关键点:

1)介绍了一种基于零存储的丢包重传机制. 它只占用了很小的内存空间和 CPU 消耗, 所以不会影响传输效率. 这种机制可以解决嵌入式系统传输多媒体文件时的丢包问题. 即便是遇到大量频繁的丢包, 系统也可以在重传后迅速恢复网络带宽, 保证流媒体传输的实时性. 同时, 它还可以解决乱序包引发的伪重传问题.

2)介绍了一种基于流媒体传输的坚持计时器, 他可以处理接收端发送的更新 RWND 的 ACK 包丢失所引发的网络崩溃和延时问题.

3)对本文所提出的协议栈优化机制进行了效率和鲁棒性测试.

4)对于想在嵌入式系统上开发流媒体功能的用户来说, 移植具有我们的优化机制的轻量级操作系统将会十分简单, 可参考 LWIP 协议栈.

参考文献

1 Zhang M, Zhao L, Tang Y, Luo JG, and Yang SQ. Large-scale live media streaming over peer-to-peer networks through global internet. Proc. of the ACM workshop on Advances in peer-to-peer multimedia streaming. ACM. 2005. 21–28.

2 Civanlar M, Cash G, Haskell B. RTP Payload Format for Bundled MPEG. 1998: 4–8.

3 Floyd S, Handley M, Padhye J, Widmer J. RFC 5348: TCP friendly rate control (TFRC). Protocol Specification, 2008: 2–5.

4 Kuschnig R, Kofler I, Hellwagner H. Improving internet video streaming performance by parallel TCP-based request-response streams. Consumer Communications and Networking Conference (CCNC), 2010 7th IEEE. IEEE. 2010. 1–5.

5 Sripanidkulchai K, Maggs B, Zhang H. An analysis of live streaming workloads on the internet. Proc. of the 4th ACM SIGCOMM Conference on Internet Measurement. ACM. 2004. 41–54.

6 Van der MJ, Sen S, Kalmanek C. Streaming video traffic: Characterization and network impact. Proc. of the Seventh International Web Content Caching and Distribution Workshop. Boulder, CO. 2002.

7 Wang B, Kurose J, Shenoy P, Towsley D. Multimedia streaming via TCP: An analytic performance study. Proc. of the 12th Annual ACM International Conference on Multimedia. ACM. 2004. 908–915.

8 张齐,劳炽元.轻量级协议栈 LWIP 的分析与改进.计算机工程与设计,2010,31(10):2169–2171.

9 Chen W, Qiu SB. The porting and implementation of light-weight TCP/IP for embedded web server. 2008 4th International Conference on Wireless Communications, Networking and Mobile Computing(WiCOM'08). IEEE, 2008. 1–4.

10 王力生,梅岩,曹南洋.轻量级嵌入式 TCP/IP 协议栈的设计. 计算机工程,2007,33(2):246–248.

11 Chiang ML, Li YC. LyraNET: A zero-copy TCP/IP protocol stack for embedded systems. Real-Time Systems, 2006, 34(1): 5–18.

12 Li J, Hao W. Research and design of embedded network video monitoring system based on Linux. Computer Science and Software Engineering, IEEE, 2008, 5: 1310–1313.

13 Perkins C, Hodson O, and Hardman V. A survey of packet loss recovery techniques for streaming audio. Network, IEEE, 1998, 12(5): 40–48.

14 Stevens WR. TCP/IP Illustrated, Volume 1 : The Protocols, 1993: 60–90.