

在低速和中速链路上的 IP 包头压缩技术

刘天钊, 黄 鑫, 阔永红

(西安电子科技大学 通信工程学院, 陕西 西安 710071)

摘 要: 在同一数据包流中, 数据包头部中大部分信息不发生变化或变化很小, 因此可以压缩这些冗余信息以提高传输效率。主要研究了在低速和中速链路上的 TCP/IPv4 和 UDP/IPv6 头部压缩方案, 用以提高带宽利用率和减小丢包率, 仿真结果证实了该算法在低中速链路上的有效性。

关键词: 头部压缩; 上下文; 二次算法; 慢开始机制

中图分类号: TP311

文献标识码: A

文章编号: 1673-629X(2007)11-0103-04

IP Packet Compression over Low or Medium Speed Link

LIU Tian-zhao, HUANG Xin, KUO Yong-hong

(School of Telecommunications Engineering, Xidian University, Xi'an 710071, China)

Abstract: The protocol headers belong to the same packet stream that are identical or with seldom change during the life of the packet stream. It is possible to compress them to improve transmission efficiency. In this paper, shown how to compress TCP/IPv4 and UDP/IPv6 headers, resulting in improved bandwidth efficiency and reduced packet loss rates over low or medium speed link. Simulation results are given to demonstrate the effectiveness of this algorithm.

Key words: headers compression; context; twice algorithm; slow-start

0 引 言

随着互联网技术的不断发展, IP 协议成为有线和无线网络通信的基础。然而, IP 协议的头部有大量的冗余信息, 降低了信道利用率。对于信道带宽有限的网络和多种复杂头部的协议的广泛使用, 问题就变得更加突出。例如, 一个基于 TCP/IPv4 协议的应用, 其基本包头长度为 40B, 当使用 IPv6 后, 包头长度增加到 60B, 而使用 mobile IPv6 协议后其包头长度增加到 100B。

如果使用头部压缩技术, 几种 IP 头部的压缩前后的对比情况如表 1 所示^[1]。

表 1 几种 IP 头部的压缩情况

头部类型	头部长度	压缩后最小 头部长度	压缩增益 (%)
IP4/TCP	40	4	90
IP4/UDP	28	1	96.4
IP6/TCP	60	4	93.3
IP6/UDP	48	3	93.75

收稿日期: 2007-01-18

作者简介: 刘天钊(1982-), 男, 山东菏泽人, 硕士研究生, 研究方向为网络通信协议; 阔永红, 副教授, 硕士生导师, 研究方向为通信信号的智能化处理。

此外, 头部压缩技术还能减少丢包率, 改善响应时间。因此, 为能有效地利用有限的带宽资源, 必须采用头部压缩技术来减小 IP 协议封装引入的额外开销。

1 基本原理

对于不同类型的数据包头部, 实现压缩的基本原理是大致相同的。

根据协议组合的特性, 对于同一报流的数据包, 一些字段用来标识该报流的特征信息, 在数据传输中保持不变, 例如 UDP 头中源端口号和目的端口号, IP 头中源地址和目的地址等字段; 另外一些字段可以通过其他字段的组合信息恢复, 例如 IP 头的校验和、IP 头长度等字段; 而其他在数据传输中不断变化的字段就是需要压缩的字段, 也就是需要传递的压缩数据信息, 包括 TCP 校验和、UDP 校验和等字段。对于前两种类型的字段, 在进行初始同步后就不需要进行传递, 后续的数据处理只需要传递变化字段的压缩信息, 这样就大大节省了带宽资源。

如图 1 所示, 头部压缩算法在压缩端和解压缩端维持一个共享的称为上下文(context)的信息。建立连接后, 发送方把原始包头存在上下文中, 以后每发一个包(不论是否压缩), 都要更新其中的包头, 以保证存储

的是该连接最后发送的包头。接收方也一样,开始把未压缩的包保存在上下文中,每接收一个数据包都要更新保存的包头,这样才能保证收发双方的同步。

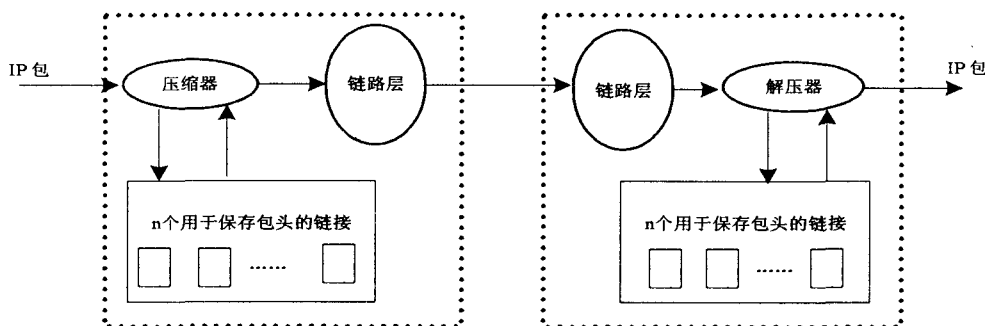


图 1 压缩/解压缩模式

2 TCP/IPv4 包头压缩方案

2.1 TCP/IPv4 包头

如图 2 所示,对于同一个流的数据包,其中的灰色部分是保持不变的,剩下的字段中,总长度(total length)字段是冗余的,可以从链路层协议中得到;头检验和(header checksum)是用以保护 IP 头部的,在 IP 头部压缩后头部已经改变,这一字段在压缩处理后可以丢弃^[2]。对于其他变化的字段,压缩后格式如图 3 所示^[3]。

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36
Protocol Version		Header Length		Type of Service				Total Length																												
Packet ID																DM		FF		Fragment Offset																
Time to Live				Protocol				Header Checksum																												
Source Address																Address																				
Destination Address																Address																				
Source Port																Destination Port																				
Sequence Number																																				
Acknowledgment Number																																				
Data Offset				100		ack		push		rst		syn		fin		Window																				
Checksum																Urgent Pointer																				

图 2 TCP/IPv4 包头

CID							
R	O	I	P	S	A	W	U
TCP Checksum							
RANDOM fields, if any							
R - octet							
Urgent Pointer Value							
Window Delta							
Acknowledgment Number Delta							
Sequence Number Delta							
IPv4 Identification Delta							
Options							

图 3 压缩 TCP 包头

开始压缩前需要用全头数据包(FULL HEAD-

ER)来初始化上下文,压缩开始后也需要不断发送全头数据包来更新解压缩端的上下文,维护两端上下文的一致性,以确保数据包正确压缩解压缩。该数据包

由未压缩的数据包组成,对于 TCP/IP 全头数据包,只是把 IP 头的长度字段由压缩上下文的 CID 值和数据包序列号(支持重排序功能)代替,其余部分不发生变化^[3]。

2.2 快速修复机制

由于 TCP 头部压缩采用差分编码,一个包的丢失使后面的许多压缩包因无法正确解压缩而被丢弃。为了提高 TCP 包的通过量,保证解压缩的成功率,提出了两种快速修复上下文(context)的机制:二次算法(The "twice" algorithm)和全头请求(Header Re-quester)。

2.2.1 二次算法

二次算法通过二次计算 TCP 检验和的方法来进行上下文修复。解压缩端计算 TCP 检验和以确定它的上下文是否已经被正确更新了。如果检验失败,则证明丢失数据包导致上下文未被正确更新,假定丢失的数据包的 DELTA 与当前值相同,利用当前压缩包的动态字段增量再次更新上下文;解压缩并再计算 TCP 检验和,以确定是否修复成功。

通过对大量 TCP 数据包的分析,在数据流(data stream)中应用两次算法恢复能达到 83% ~ 99%;但对于确认流(acknowledgment stream)成功率较低^[3]。主要由于 TCP 的延时确认机制,即依靠 TCP/IP 协议层发现数据包解压缩失败而重传可能需要一段时间,在这段时间内可能会有大量的压缩数据包由于上下文的错误而无法正确解压缩。因此,关于确认流又引入了全头请求机制。

2.2.2 全头请求机制

全头请求机制,即在解压缩端发现数据包解压缩失败时主动向压缩端发起上下文更新的请求而不依赖 TCP/IP 的保证机制。当数据传输过程中由于链路丢失包或者误码导致解压缩失败时,解压缩端会标识当前上下文无法使用,从而向压缩端发送上下文更新请求数据包(CONTEXT-STATE)要求压缩端更新该上下文。第一个字节是类型码,允许其它压缩协议共享此类型的包,当用于 TCP 时,取值为 3,其余部分是一个字节的 CID 计数器和 CID 序列^[3]。

压缩端收到解压缩端发送的上下文请求后则发送

非增量压缩 TCP 数据包 (COMPRESSED_TCP_NODELTA) 或全头数据包 (FULL_HEADER) 更新解压缩端的上下文。解压缩端通过接收数据包刷新受损的解压缩上下文,使接收端的上下文与压缩端保持一致。当链路上丢包率高的情况下压缩端可以选择只发送 COMPRESSED_TCP_NODELTA 这种类型的包头。与压缩数据包不同,该数据包携带数据包头中完整的动态变化字段,包括上下文的 CID 值、RANDOM 字段(携带 IP_ID 信息)、TCP 包头中除过端口号之外的其它所有字段以及原始数据包的数据。

然后再继续发送压缩数据包,从而保证后续压缩数据包的正确解压缩,降低了数据传输过程中的解压缩失败率。此外当有 CONTEXT_STATE 包丢失时不会影响 TCP 头部压缩的正确操作,而会发送一个新的 CONTEXT_STATE 包或等到 TCP 超时再重发。全头请求的优点是在丢失链路上往返一次就能修复 TCP 确认流,这样避免了 TCP 超时和不必要的重传。

这种数据包只用在传输 TCP/IP 数据包的情况下,并且一个全头请求中可以携带多个损坏上下文的 CID 值。UDP/IP 数据包进行周期性上下文刷新,因此不使用该类型数据包。

2.3 实现过程

(1)压缩端:收到数据包后,检验包类型并与上下文中保存的信息相比较,以发送不同类型的数据包,其流程图如图 4 所示。

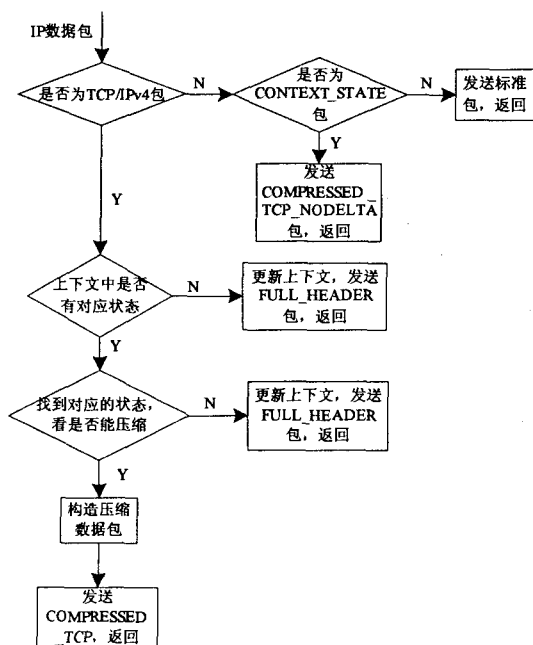


图 4 压缩端工作流程图

(2)解压缩端:收到数据包后,根据不同的包类型分别进行处理,还原为原始数据包,其流程图如图 5 所示。

示。

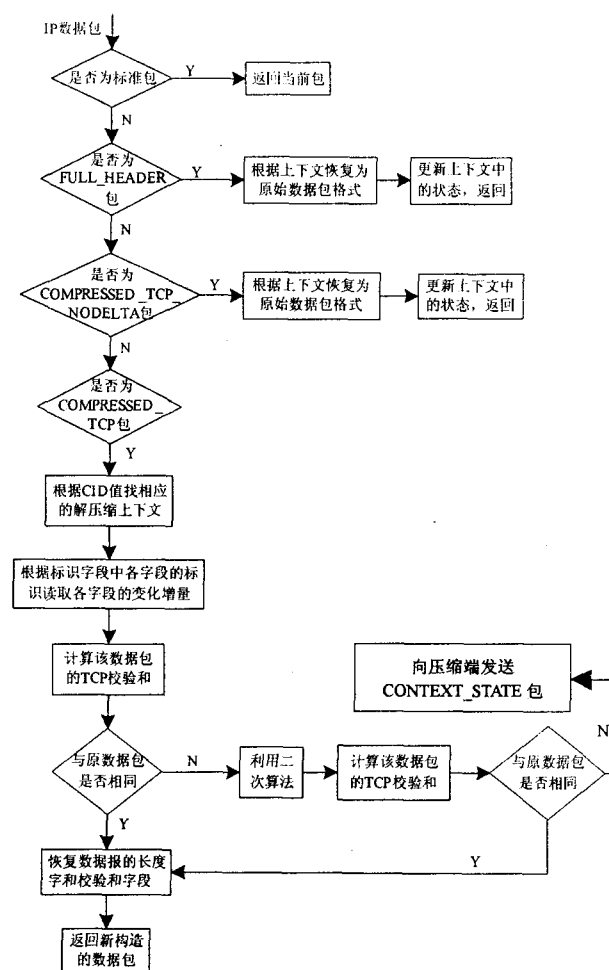


图 5 解压缩端工作流程图

3 UDP/IPv6 包头压缩方案

如图 6 所示,对于同一个流的数据包,其中的灰色部分是保持不变的。剩下的字段中,有效载荷长度(payload length)和 UDP 数据包的长度(length)都可以从链路层协议中得到。UDP 中的 checksum 是由头部,

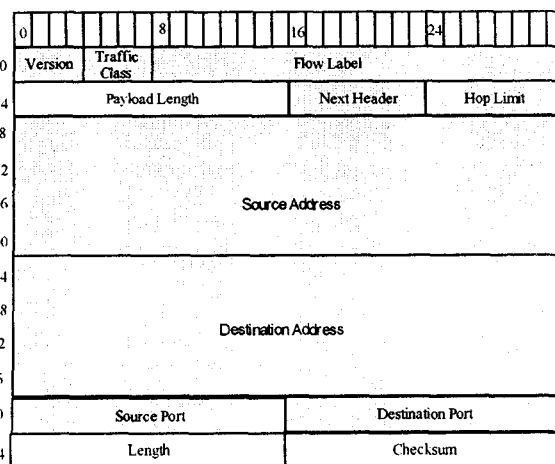


图 6 UDP/IPv6 包头

伪头部和数据共同计算得到的,每个包都会发生变化、所以不能做压缩处理。压缩包格式如图 7 所示。

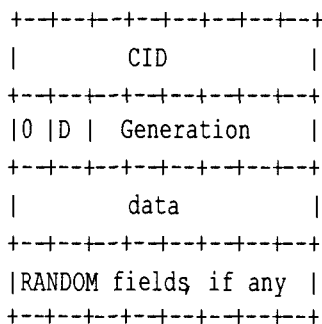


图 7 8bit CID 的 UDP 压缩包头

同样,它也需要用全头数据包(FULL-HEADER)来初始化或者更新解压缩端的上下文,确保数据包正确压缩解压缩。该数据包由未压缩的数据包组成,与 TCP 数据包不同,IP 头的长度字段则由压缩上下文 CID 和 Generation 代替。

其实现过程与 TCP/IPv4 大致相同,在此只介绍它引入的两种不同之处。

(1)当一个全头数据包丢失时,不能正常地更新解压缩端的上下文,从而导致解压缩错误。但由于 UDP 的检验和中不包含像 TCP 中序列号(sequence number)等信息,不一定能识别出此错误。故在全头和压缩数据包中都引入 Generation 来标示上下文^[3]。Generation 占 6bit,可以代表 64 个变化;只有当上下文更新后,Generation 值才发生改变。解压缩端比较压缩数据包与上下文中的 Generation 值,若不同,证明上下文未被更新,则丢弃此包或存储起来直到一个全头数据包建立了正确的上下文关系。

(2)在一个全头数据包丢失后,为了避免过长时间的丢弃包,发送全头数据包更新的间隔应当尽量小;然而,为了达到较高的压缩率,更新的间隔又应当尽量大些。为了平衡这两种要求,引入了“Slow-Start”机制^[3]。

压缩开始后,压缩端按指数增量周期发送全头数据包来刷新压缩端的上下文。

同时,为了避免过多的包被丢弃,设定了在两个全头数据包间能发送压缩数据包的上限制 F_MAX_PREIOD(默认值为 256);为了防止低速率传输过程中上下文变更导致报流中断连接,又设定了发送两个全头数据包的最大间隔 F_MAX_TIME(默认值为 5s);当超过了两个上限中的任意一个时,就强制发送全头数据包来保证解压缩的正确性。

上面只介绍了 IPv6 基本头部和 UDP 的情况,此外 IPv6 还有六种可选用的扩展头部^[4],根据具体情况

也可以压缩。由于篇幅有限,就不做详细介绍了。

4 仿真实现与性能分析

4.1 仿真环境

在 VC++ 6.0 下安装 Winpcap 来实现仿真环境,并通过软件 IRIS v4.07.1 实时监控以验证仿真的正确性。

Winpcap(Windows packet capture)是 UNIX 下的 Libpcap 移植到 Windows 下的产物,是 Windows 平台下一个免费、公共的网络访问系统,它为 Win32 应用程序提供访问网络底层的能力。它提供了以下的各项功能:

- 1) 捕获原始数据包,包括在共享网络上各主机发送/接收的以及相互之间交换的数据包;
- 2) 在数据包发往应用程序之前,按照自定义的规则将某些特殊的数据包过滤掉;
- 3) 在网络上发送原始的数据包;
- 4) 收集网络通信过程中的统计信息。

Winpcap 安装过程:

- (1) 下载 Winpcap 的安装包和程序员开发包;
- (2) 执行安装包;
- (3) 解压开发包,在 VC 的 option 的 include 和 lib 中加入 winpcap 的 include 和 lib;
- (4) 在工程的 setting 中导入 wpcap.lib 和 packet.lib 库。

4.2 性能分析

表 2 记录了 5 次在 60s 内收发 IPv4 数据包的实验结果。

表 2 实验数据(60s)

次数	总数据包数 (个)	被压缩的数据 包数(个)	压缩量 (B)	压缩率 (%)
1	1903	TCP: 862 UDP: 94	32473	87.50
2	691	TCP: 389 UDP: 27	11977	73.41
3	2141	TCP: 767 UDP: 113	30166	89.13
4	1656	TCP: 727 UDP: 18	25639	86.67
5	1502	TCP: 644 UDP: 44	23429	86.80

其中压缩率是压缩量与被压缩的数据包的原始头部大小的比值。实验结果表明,由于非 IP 包和全头数据包的存在,并不能对绝大部分的数据包都进行压缩;但对于可压缩的数据包压缩前后数据包长度明显减小。所以采用此压缩方案能够很好地提高传输效率。

5 结束语

介绍了数据包头压缩方案,并在 VC++ 6.0 环境

(下转第 186 页)

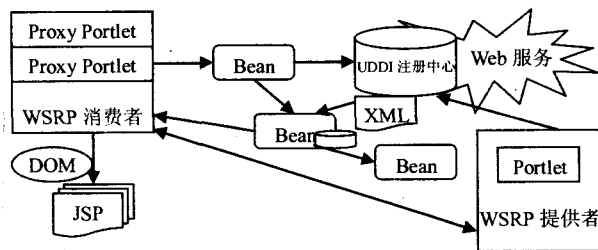


图 4 Portlet 和 WSRP 应用程序流程图

些对象同样可以返回给持久层,从而达到数据库里数据更新。在该层建立 XML 映射文件来管理持久性对象和保护值的对象,使之与数据逻辑和业务逻辑分离和增强数据访问的安全性,从而提高逻辑访问的松散耦合度,顺利完成 BO 在集成层中控制完成分布式数据交换。并在该层实现相关采购招标规定约束,使采购招标工作顺利进行。如图 5 所示是整个通用采购招标系统层次逻辑图。

3 总结

从目前的采购招标系统现状开始,提出基于 Web 的采购招标系统的通用框架设计研究,确立该通用框架采用 Struts Portlet, Direct Web Remoting, Spring 和 Hibernate 四款开源轻量级框架,并使用 Web 服务、Ajax 和 Web Services for Remote Portlet 等标准、技术和规范在 J2EE 和 Ajax 技术下采用层次结构集成该框架,对组成的层次功能和方法进行了详细的分析,这样大大降低了开发和数据处理难度。

参考文献:

- [1] 殷茗,赵嵩正. 基于 Web 的采购招标系统的分析与设计[J]. 计算机应用与软件, 2006(2): 50-52.
- [2] McCarthy P. Ajax for Java developers: Build dynamic Java applications[EB/OL]. 2005-09. http://www-128.ibm.com/dev-eloperworks/java/library/j-ajax1/?S_TACT=105AGX52&S_CMP=cn-a-j.

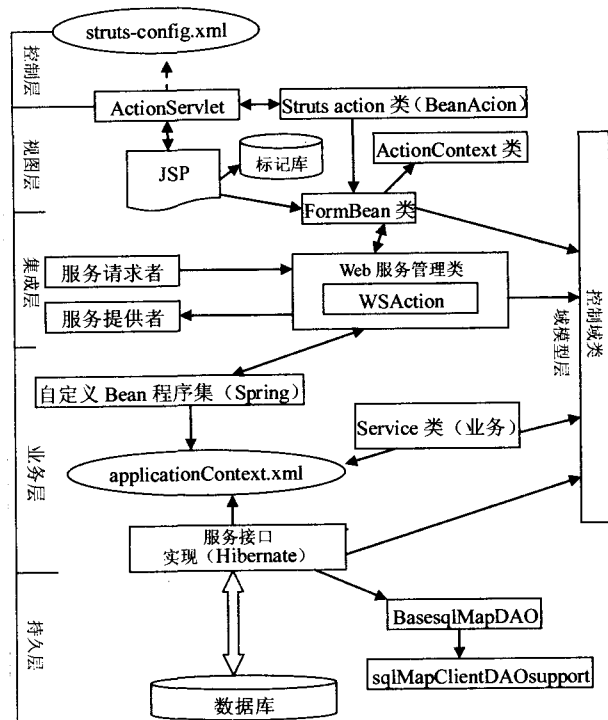


图 5 通用采购招标系统层次逻辑图

com/dev-eloperworks/java/library/j-ajax1/?S_TACT=105AGX52&S_CMP=cn-a-j.

- [3] Walker J, Goodwin M. Direct Web Remoting[EB/OL]. 2006. <http://getahead.ltd.uk/dwr/ajax>.
- [4] Hibernate Reference Documentation Version: 3.2.0. ga[R/OL]. 2006. <http://www.hibernate.org>.
- [5] 周彩兰,李素芬,孙琳. Hibernate 在 Spring 中的研究与应用[J]. 计算机技术与发展, 2006, 16(10): 62-67.
- [6] Johnson R, Hoeller J, Arendsen A, et al. Spring2.0[R/OL]. 2006. <http://www.springframework.org>.
- [7] Abdelnur A, Hepper S. JSR-168[S]. Java™ Portlet Specification. Sun Microsystems, Inc, 2003.
- [8] 郁雪,常鹏,董旭源. 基于 Struts 框架的高校网络教务信息系统设计[J]. 电子技术应用, 2006(2): 30-33.

(上接第 106 页)

下实现仿真。实验结果表明,此方案能够达到较高的压缩率,提高了信道利用率;同时由于压缩头部的每个包传送的比特数比较少,对于有固定的比特差错率的链路,压缩头部比未压缩的丢包率低,从而提高了传输效率。这种算法适用于低中链路的情况下,对于误码率较高($10E-3$ 或更大)、延迟时间长(来回响应时间 $100 \sim 200ms$)的无线链路,性能就会有所下降^[5]。

参考文献:

- [1] EFFNET A B. An introduction to IP header compression[EB/OL]. 2004. <http://www.ietf.net.com>.

- [2] Jacobson V. Compressing TCP/IP Headers for Low-Speed Serial Links[EB/OL]. Request For Comment 1144, 1990-02. <http://www.ietf.org>.
- [3] Degermark M, Nordgren B, Pink S. IP Header Compression[EB/OL]. Request For Comment 2507, 1999. <http://www.ietf.org>.
- [4] Eeering S, Hinden R. Internet protocol[EB/OL]. version 6. Request For Comment 2460, 1998. <http://www.ietf.org>.
- [5] Degermark M, Engan M, Nordgren B, et al. Low-loss TCP/IP header compression for wireless networks[M]. [s.l.]: ACM MobiCom, 1996.