μReTi：小尺寸实时嵌入式系统的 TCP/IP 精简化研究

周海鹰，左德承
（哈尔滨工业大学计算机科学与技术学院，哈尔滨 150001）

摘要：通过分析 TCP/IP 协议的不同简化实现，提出 TCP/IP 协议的精简策略与设计准则。采用基于模块化的跨层设计思想，实现一套 TCP/IP 协议简化版本—μReTi。μReTi 采用静态的内存预分配与管理机制优化系统数据结构，采用宽松的分层机制以及“后门”技术减少数据复制并重用共享内存。通过在不同架构感知节点上移植 μReTi，评测分析 μReTi 的性能，验证其低资源消耗的特点。

关键词：无线感知设备；TCP/IP 协议精简；μReTi；模块化跨层设计；低资源消耗
中图分类号：TP393.04

μReTi：Minimized Real-time Embedded TCP/IP Research

ZHOU Haiying, ZUO Decheng
(School of Computer Science and Technology, Harbin Institute of Technology, Harbin 150001)

Abstract: By analyzing and concluding the different simplified implementations of TCP/IP protocols, this paper proposes the TCP/IP simplified strategies, approaches and design criteria. Based on the modular cross-layer design approach, this paper introduces a simplified TCP/IP version – μReTi, which adopts the static memory pre-allocation and management mechanism to optimize data structure, and the relaxed layered scheme and the ‘wickets’ technique to avoid data copying and reuse shared memory. μReTi has been ported on two processor architectures, and the experiential results prove it has low resource consumption.

Keywords: Wireless sensor device; TCP/IP protocol simplified; μReTi; Modular cross-layer design; Low resource consumption

0 引言

基于伯克利 TCP/IP 协议规范[1]，传统的 TCP/IP 实现能够运行在具有不同带宽与误差率的各种互联技术上。但是，传统技术更多专注于在不同网络环境下通信的兼容性问题而非协议的资源消耗问题，运行一个完全 TCP/IP 实现至少需要数百 KB 的代码和数据内存。嵌入式感知节点为控制成本，具有典型的低资源特点，内存与计算资源有限。因此，考虑到实时性和低资源消耗的嵌入式应用需求，移植传统协议簇不是一个很好的选择[2]。

目前，尽管已有各种针对嵌入式系统的 TCP/IP 实现，但人们对于 TCP/IP 协议簇简化机制的研究仍缺乏足够的重视。本文通过分析 TCP/IP 协议簇的通信机制，提出了基于模块化跨层设计的协议精简策略与方法，开发了一套适合于资源受限的嵌入式感知节点的 TCP/IP 协议簇的简化实现版本，μReTi（Minimize Real-time and Embedded TCP/IP implementation：最小化实时嵌入式 TCP/IP 实现）。本文第一节提出了 TCP/IP 协议簇的精简策略与方法；第二节介绍了 μReTi 设计与实现，包括着主要功能模块与数据结构，算法机制以及通用的控制流程等；第三节给出了 μReTi 的性能评估与比较；最后一节总结了当前工作，提出了下一步

基金项目：部委预研重点基金（N°.9140A16020309HT01），教育部博士点基金项目（N°.200802131024），哈尔滨市科技创新人才研究专项基金（N°.2009RFLXG009），科技部国际科技合作计划（N°.2010DFA14400）。
作者简介：周海鹰，（1975-），男，副教授，主要研究方向：计算机系统结构、移动计算技术、无线自组网技术、容错计算等。 E-mail: haiyingzhou@hit.edu.cn

研究方向。

1 TCP/IP 实现的精简策略

BSD TCP/IP 是一个多层次网络协议的复杂集合，集成了不同网络标准与技术，能适用于不同的网络环境，但同时其复杂的实现也带来了巨大的资源开销，因不能直接应用于嵌入式系统。因此，设计一个简化 TCP/IP 实现的关键就是根据实际环境需求简化 BSD 实现，降低其资源需求[3]。虽然众多版本的 TCP/IP 实现有各种不同的简化模式，本文基于针对无线感知网应用环境的一些假设，总结和提出了 TCP/IP 实现所遵循的通用简化规则：

首先，假设在嵌入式系统的远程对等端运行着一个具有完全功能、标准兼容的 TCP/IP 实现。依靠远程端主机的完全协议实现，即使在嵌入式系统上运行一个简化、标准不兼容的 TCP/IP 实现依然能保证两者间的通信。比较常用的简化方法是裁剪 TCP/IP 协议簇以及移除某些通信机制等，降低 TCP/IP 实现的复杂性，减少内存需求。代表性实现，移植在 PIC 12C509A 芯片上的 iPic Web 服务器[4]（仅 256bits），由于未实现 UDP 协议及 TCP 协议模块的重传机制，此 Web 应用不能获得 TCP 激活连接信息，每次数据传输完成，TCP 连接必须关闭；另一个例子是 Atmel 的 TCP/IP 实现[5]，其移除了 TCP 冗余控制机制。

其次，假设嵌入式系统每次只能支持一个或少数几个网络连接。考虑到多数嵌入式网络应用仅仅针对一个特定的网络业务，对于资源紧张的嵌入式系统，此种假设是很多情况下成立。基于此假设的 TCP/IP 实现，同样能够在很大程度上降低其资源消耗。例如，德州仪器公司提供的基于 MSP430 的 TCP/IP 实现[6]能支持一次一个 TCP 连接网络的 Web 服务器。

第三，取代传统的基于全局堆的动态分配帧缓存的方法（malloc），采用静态的帧缓存预分配机制。一方面，如果移除协议头部参数，大多数 TCP/IP 协议头部（IP/UDP/TCP 等）都具有固定的尺寸；另一方面，在一些应用中，数据帧长度是固定的或预先设定的，确定不超过 MTU（Maximum Transfer Unit：最大传输单元）和 MRU（Maximum Receive Unit：最大接受单元），因此，采用静态帧缓存预分配方法可以简化 TCP/IP 实现的内存管理机制，提高其性能，尤其是针对具有固定数据长度的网络业务。例如，A. Dunkles[7]开发的 *µIP* 实现给出了静态预配置帧缓存的一个实例，其中，TCP 监听端口数，TCP 连接以及 ARP（Address Resolution Protocol：地址解析协议）的表项目都是预先配置的。

第四，针对某些特定应用，IP 分帧与重组模块是可选的并能够被移除。分帧与重组模块是内存管理机制中最复杂的部分，具有重组 IP 分帧包功能的 TCP/IP 实现通常需要相当多的帧缓存以存储分帧包。如果特定的应用能确保没有分帧数据出现在嵌入式系统端，重组函数能够被移除；而如果传输数据的大小被限定在 MTU 的范围内，分帧函数就能够被移除。

最后，由于在实时嵌入式系统中执行数据复制的代价大，因此应尽可能减少数据复制以降低相应开销。通常方法是将数据包能够被存储在一个特定缓存区，数据通过指针而非复制的方式在 TCP/IP 协议各层之间传递。同时，一些 TCP/IP 实现扩展了 BSD 的 MBUF 机制以允许数据在用户空间和内核空间共享。例如，InterNiche Technologies 公司宣称它们的 NicheLite[8]和 NicheStack[9]实现能支持“零”数据复制，从而提供非常高的系统性能。

基于上述精简假定与策略，针对无线感知网应用，本文提出了面向感知节点的 TCP/IP 的精简实现—µReTi，主要简化设计思想与方法：1) 采用模块化设计，针对不同感知应用，选择性地加、卸载协议功能模块；2) 采用跨层设计方法，扩展 BSD 消息机制，支持数据在用户空间和内核空间共享，减少数据复制次数以降低内存开销；3) 采用静态的帧缓存预分配机制，简化 BSD 缓存管理机制。

2 μReTi 设计和实现

μReTi 最初设计意图是为基于感知网的远程心电实时看护系统（STAR）[10]的智能无线心电图传感器节点提供网络互联功能。STAR 应用具有受限的网络连接（同一时间只有一个 TCP 连接和一个 UDP 连接），并具有唯一的物理媒介（有线或无线的串口链路）。收发数据包的数据尺寸可预测，分别不超过 MRU 和 MTU。同时，远程通信端是标准 PC，运行完全、标准兼容的 TCP/IP 实现。此类应用条件符合 TCP/IP 实现的上述简化假设。

μReTi 设计采用基于模块化的跨层协议设计思想，提供“后门”以方便模块之间的数据交换。μReTi 的主要数据结构与消息处理机制分别在小节 2.1 与 2.2 中介绍，组成 μReTi 的五个核心协议模块（PPP/IP/ICMP/UDP/TCP）以及应用层协议在小节 2.3 中描述。

2.1 μReTi 数据结构

缓存子系统负责协议实现的存储管理。μReTi 采用静态预分配的缓存机制分配数据包缓存、网络接口表、路由表以及 socket 接口等存储空间。为优化数据存储并简化协议实现，针对 TCP/IP 协议主要缓存单元，μReTi 定义与优化了其数据结构：数据包存储结构（new_packet）、网络接口结构（new_ndevice）、路由表结构（new_route）以及 socket 结构（new_socket）。

2.1.1 数据包存储结构—new_packet

图 1 显示了数据包存储结构—new_packet。考虑 STAR 应用网络连接的特点，μReTi 只需同时支持两个在线网络连接（TCP 与 UDP），且数据包大小固定，不需进行分帧重组处理。因此，μReTi 静态预分配四个 new_packet 数据成员用于支持两个在线网络连接的收发操作。new_packet 数据结构由三部分组成：

New_packet

```
Struct new_packet
{
    struct  IPHDR    iph;                    /*IP head*/
    union{
            struct  UDPHDR    udph;         /*UDP  head*/
            struct  ICMPHDR   icmph;        /*ICMP head*/
            struct  TCPHDR    tcph;         /*TCP  head*/
    }trah;
    unsigned char *  dat_ptr;                /*data pointer*/
    short            dat_len;                /*data size*/
    short            ipp_len;                /*ip packet size*/
    unsigned char    flag;
    /* 0000                                  /*relative socket index*/
            0                                /*unused*/
              1                              /*transport layer head use flag*/
                0                            /*ippbuf unit use flag*/
                  1*/                        /*packet type flag*/
};
```

图 1 μReTi 数据结构: new_packet

协议头域：*iph* 是 20bits 的 IP 数据包头部信息（μReTi 不支持 IP 头部参数）。*trah* 封装了三种不同传输层协议（TCP/UDP/ICMP）数据报头部信息：*tcph* 是 20bits 的 TCP 数据段头（在 μReTi 中，TCP 头部参数作为传输数据保存在 TCP 数据段）；*udph* 是 8bits 的 UDP 数据报头；*icmph* 是 8bits 的 ICMP 协议数据包头部（μReTi 将 ICMP 作为传输层协议实现）。

数据域：*dat_ptr* 与 *dat_len* 分别指示 IP 数据包的数据缓存地址与数据大小，传输数据可位于用户空间或内核空间以减少数据复制。*ipp_len* 指示 IP 数据包的长度，表示为 *dat_len*、

iph 头部长度以及 *trah* 头部长度之和（在 µReTi 应用中，*ipp_len* 应小于 MTU 或是 MRU，以避免执行数据包的分帧重组操作）。

标识域：*flag* 是 IP 数据包标识符，用于指示数据包各种状态：前四个 bit 位指示了 IP 包的关联 socket 在 socket 表中的索引值；第五个 bit 位暂时保留；第六个 bit 位显示当前 IP 包是否包含有传输层协议；第七个 bit 位显示当前 *ippacke* 缓存是否被使用或是闲置；最后一个 bit 位指示了 IP 包的类型（0：输入包；1：输出包）。

2.1.2 网络接口结构—new_ndevice

µReTi 的网络接口结构如图 2 所示。µReTi 支持两类网络接口：串行链路以及回送界面（loopback），因此，两个 new_ndevice 数据成员被静态预分配用以存储网络接口。Index 域变量用来显示界面成员在网络接口队列中的索引。name 域存储网络接口类型的英文缩写，如 sl（串行链路，serial line）和 lb（回路，loopback）。ip_addr，ip_broadaddr 和 ip_mask 表示网络接口的 IP 地址标识。回路界面的 IP 地址设为‘127.0.0.1’，串行链路的 IP 地址被远程 PPP 服务器动态分配或由用户手动设置。if_output 和 if_input 为网络接口的输出、输入函数指针，网络接口调用两个接口函数实现与物理媒介的交互。

New_ndevice

```
struct new_ndevice
{
    unsigned char       index;
    unsigned char       name[2];
    struct ip_addr      if_addr;
    struct ip_addr      if_broadaddr;
    struct ip_addr      if_mask;
    int                 (*if_output)(struct ppp_state *state,int ipp);
    void                (*if_input)(unsigned char *buf, int len);
};
```

图 2 µReTi 网络接口结构: new_ndevice

2.1.3 路由表结构—new_route

µReTi 的 IP 协议模块定义了如图 3 所示的路由表结构用于存储路由信息：ro_dst，ro_src，ro_mask 和 ro_gate 域分别描述了目的主机 IP 地址、网络接口地址（源 IP 地址）、目的网络掩码以及网关地址；ro_flag 域指示了网络路由表的特征值标识；ro_ifdex 是一个网络接口的索引指针。µReTi 静态预分配了五张路由表，每张路由表均关联有唯一的网络接口（目前，µReTi 仅支持两个网络接口：串行接口以及回路接口）。由于路由表项目少，IP 模块路由算法简化为：如果 IP 包的目的掩码地址等于路由表项中目的接口的掩码地址，对应的网络接口被选取；而如果没有发现可匹配的接口，缺省的网络接口（串行链路）将被采用。

New_route

```
struct new_route{
    unsigned long   ro_dst;    /*destination host adress or net address*/
    unsigned long   ro_src;    /*interface addr*/
    unsigned long   ro_mask;   /*destination net and subnet mask*/
    unsigned long   ro_gate;   /*gateway address*/
    unsigned short  ro_flag;   /*route flags*/
    int             ro_ifdex;  /*the index of interface device*/
};
```

图 3 μReTi 路由表结构: new_route

2.1.4 Socket 结构—New_socket

μReTi 采用与 BSD socket API 完全兼容的接口界面和调用模式，可支持在不同数据空间之间的直接读、写操作。图 4 显示了 μReTi 定义的 socket 接口数据结构。socket 地址（new_skaddr）由一个 IP 地址和端口号所组成，源和目的 socket 地址构成了一个 socket 对，用于唯一标识一个网络连接。dbuf 和 dlen 分别表示数据的地址指针以及数据大小，将被直接映射到一个被分配的 new_packet。因此，协议模块能直接管理应用数据缓存而不需要数据复制。type 标识不同的 socket 类型：SOCK_STREAM、SOCK_DGRAM、SOCK_RAW，分别对应 Protocol 域指示的协议类型：TCP、UDP、ICMP。dev_if 与 ro 域变量指示了当前连接所对应的网络接口索引以及路由表项索引。State 域指示 TCP 的连接状态，仅用于 SOCK_STREAM 界面和 TCP 协议。skfd 与 used 分别表示 socket 在 socket 队列中的索引值以及其使用状况，被用来指派新的 socket 或从队列中搜寻对应的 socket。

New_socket

```
struct new_socket
{
    int                 skfd;      /* socket index in sock queue*/
    int                 used;      /* used flag. 0: unused; 1 used*/
    struct new_skaddr   srcskaddr; /* source socket-address*/
    struct new_skaddr   dstskaddr; /* destination socket-address*/
    unsigned char   *   dbuf;      /* data buffer pointer*/
    int                 dlen;      /* data length*/
    int                 type;      /* socket type: DGRAM,STREAM,RAW*/
    int                 prot;      /* transport protocol: udp, tcp, raw*/
    int                 dev_if;    /* device interface index*/
    struct new_route *  ro;        /* route table pointer*/
    unsigned char       state;     /* tcp socket state*/
};
```

图 4 μReTi Socket 接口结构：new_socket

2.2 μReTi 消息处理机制

消息子系统揭示了不同协议模块之间网络功能的消息处理机制。针对系统协议的模块化设计，μReTi 采用宽松的分层机制以及“后门”技术能够减少数据复制并且大量重用共享内存。μReTi 的数据流图描述了数据复制以及内存使用的情况，如图 5 所示。

对于消息发送，μReTi 仅需执行一次数据复制。发送的数据直接从应用程序复制到网络

输出队列。对于消息接收，根据不同的消息类型，μReTi 执行一次或两次的数据复制：如果接收的消息是 PPP 配置帧、ICMP 数据包或是 IP 转发包，这就意味着接收的数据是为了协议配置、或是测试（Ping 服务）或是转发目的，数据直接复制到网络输出队列；如果接收的消息是 UDP 或 TCP 数据包，数据首先从网络输入队列复制到 socket 队列，从而唤醒一个暂停的应用进程，然后由该进程将数据复制到应用进程的数据缓存以等待进一步的处理。

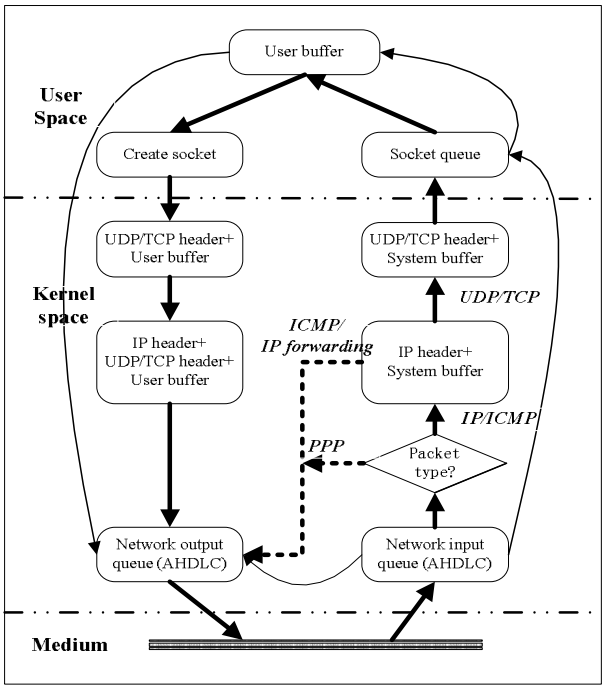

图 5 μReTi 数据流图

2.3 μReTi 协议模块

μReTi 系统包括五个核心协议模块：PPP/IP/ICMP/UDP/TCP。图 6 显示了 μReTi 系统的函数调用层次。

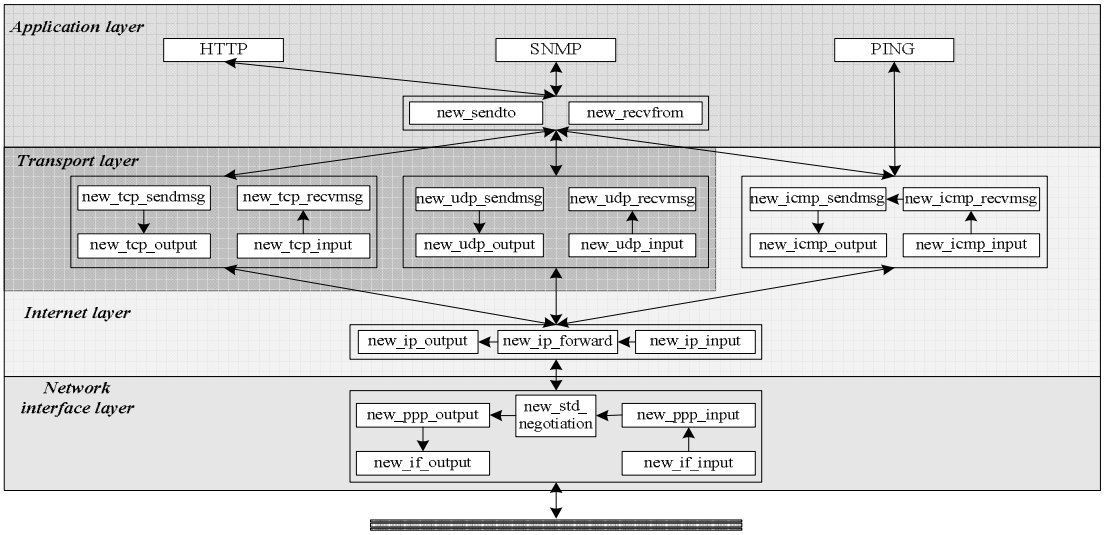

图 6 μReTi 函数调用模型

2.3.1 网络接口层：PPP 模块

μReTi 实现了简化的 PPP（Point-to-Point Protocol）协议模块为嵌入式感知设备提供基于点对点的网络互联能力。当作为 PPP 服务器的对等端机上运行采用完全、标准兼容的 PPP 协议时，一个精简的、标准不兼容的 PPP 实现能够在嵌入式节点设备上运行。μReTi 实现一个简化版的 PPP（Point-to-Point）协议模块，为嵌入式设备提供基于串行链路的点对点网络连接能力。μReTi 支持三种 PPP 帧类型：IP 数据包、链路控制协议（LCP）数据、IP 网络控制协议（IPCP）数据。

PPP 协议模块的核心是一个简化版的 PPP 有限状态机，由阶段（phase）、事件（event）、动作（action）和状态（state）转移来定义实现，用于协商、配置和管理串行链路。考虑到主动模式是 STAR 系统最常用的工作方式，由 STAR 系统用户通过与 PPP 服务器通信，主动打开和关闭一个点对点连接。因此，μReTi 实现的 PPP 有限状态机是 RFC 1661 定义的标准状态机的子集，支持主动运行模式。STAR 系统通过与 PPP 服务器主动通信，配置 LCP 和 IPCP 参数，建立 PPP 链路连接，并实现 IP 数据包的在 PPP 链路上的传输。

2.3.2 网络层：IP 模块

IP 协议模块实现基本的 IP 数据包发送、接收以及转发功能。在 μReTi 中，为避免 IP 包分帧操作，输出数据的大小被限定为不超过 MTU。对于接收包，IP 模块调用 new_ip_input()处理：首先判断 IP 包是否发往本机。若是，则调用对应的传输层协议模块接口函数进一步处理；若否，通过调用 new_ip_forward()进行转发，最后，调用 new_ip_output()通过缺省的网络接口发送 IP 包。对于发送 IP 包，IP 模块调用 new_ip_output()处理：首先，填充 IP 头部单元，计算 IP 头部校验码；其次，通过调用 new_ip_route()寻找合适的网络接口和路由；最后，调用网络接口的输出函数 *if_output* 完成数据传输。

2.3.3 传输层：ICMP 模块

ICMP 协议主要用于报告错误条件以及查询主机参数。为简化处理在，μReTi 中 ICMP 作为传输层协议被实现。目前，ICMP 模块只支持 ICMP echo 机制，也就是 ping 功能。当收到来自远端主机的 ICMP echo 请求包时，自动调用 new_icmp_input()响应，通过交换源与目的 IP 地址，改变 echo 消息类型以及重新计算 ICMP 校验和，产生 ICMP echo 应答包并将其发送给 IP 层接口函数。

2.3.4 传输层：UDP 模块

当应用准备传输数据，UDP 输出函数 new_udp_sendmsg()首先被调用，并立即调用另一个输出函数 new_udp_output()（为与 TCP 协议统一格式，定义两个 UDP 输出函数）。此时，一个 new_packet 缓存被分配用于存储 UDP 数据报。同时，函数计算 UDP 校验和并填充 UDP 头部。UDP 伪头部能够被存储在 new_packet 缓存的 IP 头部区域以减少数据消耗。如果应用程序没有指定源 IP 地址和端口号，系统缺省的 IP 地址和端口将被指派给这个数据报并被赋值给对应的 socket 对（socket-pair）。最后，UDP 报文被发送到 IP 层，由 new_ip_output()实现传送。

当 UDP 数据报到达，UDP 输入函数 new_udp_input()被调用。此时，函数检验 UDP 校验和。new_udp_lookup()被用来从 socket 队列中寻找与接收 UDP 报文相对应的 socket 对。如果找到一个相关的 socket 对，UDP 数据将被复制到对应的 socket 的数据缓存。在下一次进程调度中，new_udp_recvmsg()将数据从 socket 缓存复制到应用程序的接收内存。随后，

被挂起的应用程序将返回并继续后续的操作。如果没有发现对应的 socket 对，当前这个 UDP 数据报只是简单的被丢弃。

2.3.5 传输层：TCP 模块

μReTi 的 TCP 实现是一个标准 TCP 协议的子集，实现了 TCP 服务器的基本功能：被动的打开 TCP 连接、TCP 数据段传输，以及主动或被动的关闭 TCP 连接。μReTi TCP 模块通信机制如下：

发送数据段

接口函数调用 new_tcp_sendmsg()用于处理 TCP 数据传输。数据被分割为合适尺寸的多个小块（尺寸小于 MTU-28），并被赋予按次序排列的序列号。随后，在一个控制循环里，调用 new_tcp_output()传输每一个数据单元。new_tcp_output()从 new_packet 队列中选择一个空闲成员用以存储 TCP 数据段。TCP 头部被填充，校验和被计算。SYN 数据段将被单独的处理。SYN 数据段并不包含数据但可能具有 TCP 参数（比如，MSS）。因为在 new_packet 结构中 TCP 头部具有固定的尺寸，TCP 头部参数因此不得不存储在数据区域，但它必须作为头部的一部分被处理。像 UDP 输入函数一样，在传输数据到 new_ip_output()以前，IP 地址和端主要口号必须被指派到这个数据段并被赋值给对应的 socket 对。

接收数据段

IP 模块调用 new_tcp_input()函数处理接收的 TCP 数据段。在这个函数中，TCP 校验和被用来验证数据。new_tcp_lookup()被调用从 socket 队列里查找合适 socket 对。当找到一个 socket 对时，针对不同的数据段类型，不同的处理将被执行。如果接收的是一个 SYN 数据段、FIN 数据段、或是一个针对 SYN 及 FIN 的 ACK 数据段，则 TCP 服务器将被动的打开和关闭一个 TCP 连接。而如果接收的是数据，则这个数据将被直接复制到对应的 socket 缓存。在下一次的进程调度中，new_tcp_recvmsg()将数据从 socket 缓存复制到应用程序的接收内存。随后，被挂起的应用程序将返回，并继续后续的操作。

滑动窗口

STAR 应用系统只有一个 TCP 应用（Web 服务器），并在同一时间只有一个 TCP 连接。对于 Web 服务器应用，HTML 页面被存储在预分配的数据缓存。当用户通过 Internet 浏览器请求 HTML 网页，一个 TCP 连接将在用户和 Web 服务器之间被建立。new_tcp_sendmsg()函数被调用将 HTML 页面缓存划分为几个数据小块。因此 Web 服务具有多个 TCP 数据段需要被传输。这些 TCP 数据段能够被连续传送而不必等待每一数据段的应答。μReTi 实现了滑动窗口机制，并且仅需要几行管理 HTML 页面缓存的代码，因此并没有增加 μReTi 实现的复杂性。

定时器和重传

μReTi 仅实现了 TCP 服务器功能，因此针对每一个连接，我们的 TCP 实现仅需要维护三个定时器：重传定时器，延时定时器，以及 2MSL 定时器。这三类定时器由系统硬件定时器所驱动。每一次硬件定时器被激活，这三类 TCP 定时器时间变量将递减。一旦定时器时间变量递减为零，一些特定的动作将被激活。μReTi 定义的缺省重传时间是 2.84s（TMS320C5410 处理器 200 个时钟滴答），缺省延时时间是 5.68s，两倍于重传时间，缺省 2MSL 时长为 30s。

由于 TCP 应用的数据缓存是预分配的并且总是保持就绪状态。因此，μReTi 的数据重传不需要重新分配和释放数据缓存空间，同时也不必构造新的 TCP 数据段。事实上，TCP

数据重传与正常数据传输相类似，TCP 实现的复杂性并没有因此而增加。

RTT（Round-Trip Time）时间估计

在 μReTi 中，Ping 程序（ICMP echo 消息）可以被用于评测 RTT 时间。但由于 STAR 系统的 TCP 应用非常简单，并且实际的物理媒介是唯一的（点对点的串行链路），三类 TCP 定时器的时长以及网络流量可以被预测和预设定。因此，STAR 系统并不需要通过测量 RTT 时间来评估定时器的时长。

流量控制和拥塞控制

TCP 流量控制机制的目的在于允许主机间以广泛可变的尺寸大小通信。TCP 流量控制能够通过改变 TCP 头部的 window size 域来实现。这个变量显示了发送端可以获得的用于 TCP 通信的缓存大小。μReTi 目前没有集成 TCP 流量控制机制，window size 域变量被设定为一个恒定常数。原因之一在于，μReTi 不支持动态数据分配，网络输入/输出队列的大小决定了 TCP 数据段的尺寸；另一原因在于 STAR 系统的 TCP 应用具有很少的 TCP 数据交换，且每次仅有一次 TCP 连接被允许。因此，针对当前的 STAR 系统，采用流量控制机制并不能实际提高系统性能。

通过限制在网络上同时传输的 TCP 数据段的数目，TCP 拥塞控制机制能改变发送端的网络速率以适应网络容量的变化，并试图避免潜在的网络拥塞。由于 STAR 系统的 TCP 应用仅支持同一时间的一个 TCP 连接，这样就不存在着潜在的网络拥塞条件，因此网络拥塞控制机制同样并没有被集成到当前的 μReTi 中。

2.3.6 应用接口层：SNMP 代理与 Web 服务器

μReTi 支持七种基本的 socket 调用，按照功能划分，可以被划分为三类：1) socket 分配：new_socket()；2) 连接管理：new_close()，new_bin()，new_listen()，new_accept()；3) 数据发送：new_sendto()和 new_recvfrom()。

在应用协议层，作为 UDP 和 TCP 的具体实例，μReTi 包含一个 SNMP 代理和一个最小化的 Web 服务器，它们被配置成为操作系统的两个系统进程。μReTi 的最小化 Web 服务器仅实现了 HTTPver1.0 的部分功能，仅支持来自客户的‘GET’请求。包含有 STAR 应用系统状态信息以及配置信息的 Web 数据将显示在客户浏览器界面上。

μReTi 的 SNMP 代理实现了 SNMPver1 和 SNMPver2c 的部分功能，网络嵌入式系统扮演着 SNMP 代理的角色，响应管理者的请求并报告嵌入式系统的状态。SNMP 代理支持五类基本操作：GET，GETNEXT，SET，GETRESPONSE 和 TRAP。管理者发送三类请求给 SNMP 代理（UDP 端口号 161），代理则通过发送“陷阱”通知给 SNMP 管理者（UDP 端口号 162）。SNMP 代理实现了两组 MIB 变量：公有的 system 组和私有的 STAR 组。STAR 系统的错误代码，通信秘钥，以及其它配置和管理信息均可以成为 MIB 变量。通过设置和获取这些 MIB 变量，SNMP 管理者具有了远程配置和管理 STAR 系统的能力。

3 性能评估

μReTi 被分别移植到两个不同的处理器平台：数字信号处理器（TMS320C5410）与微控制器（MSP430F149）。本节分析了 μReTi 在不同硬件平台上的资源消耗，并与其它的 TCP/IP 实现进行了性能比较。

3.1 数据和代码尺寸

表 1 展示了针对 TMS320C5410 编译的 μReTi 协议模块代码与数据尺寸，图 7 显示了协

议模块的尺寸所占百分比：图 7 (a)中，由于 MIB 库需占据大量的系统内存，SNMP 具有最大尺寸。图 7 (b)中，由于 μReTi 静态预分配的一个 11×6 的二维数据矩阵用于存储 PPP 状态转移表，同时 PPP 采用两次的协商机制（LCP 和 IPCP），PPP 模块因为复杂的状态机机制占有 40%的内存开销；TCP 模块因为提供一个面向连接的、可靠的服务而具有 20%的内存开销，它在所有的传输层和网络层模块中具有最大的内存开销。

表 1 μReTi 协议模块代码与数据尺寸（TMS320C5410）

协议	代码（字）	数据（字）		数据尺寸（字）	所占百分比	
		变量	常量			
PPP	2506	768	780	4054	27%	40%
IP	1348	224	58	1630	11%	16%
ICMP	485	5	3	493	3%	5%
TCP	1988	25	21	2034	14%	20%
UDP	834	2	6	842	6%	8%
HTTP	249	3	172	424	3%	--
SNMP	3311	707	149	4167	29%	--
API(SOCKET)	414	80	0	494	3%	5%
Drivers &others (IF)	587	10	17	614	4%	6%
总计	11722	1824	1206	14752	100%	100%

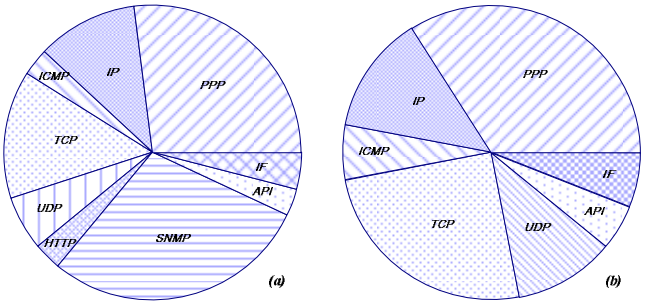

图 7 μReTi 协议模块内存占用百分比（TMS320C5410）
a) 包含应用层协议; b) 包含应用层协议

针对 TMS320C5410 芯片每字节具有 16bits 的特征，μReTi 采用了一种降低存储开销的方法：充分利用一个字节的前 8bits 空间。例如，在 TMS320C5410 中，一个‘char’用 16bits 表示，但只有 8bits 被实际使用，另外 8bits 被浪费掉。因此，使用这 8bits 的空闲资源可以节省不少内存。

表 2 展示了针对 MSP430F149 编译的协议模块的代码与数据尺寸，图 8 显示了协议模块的尺寸所占百分比。因为 MSP430F149 资源限制，两个应用层协议模块（SNMP 和 HTTP）并没有被编译进 μReTi 在 MSP430F149 的实现中。在表 2 中可以看出，每一个协议模块的代码尺寸都有显著的增加，这主要是因为 MSP430F149 并不直接支持 32 位的操作，同时不具有像 TMS320C5410 一样的利用 8 比特的空闲资源。在 MSP430F149 中，一个 32 位的操作将被编译器扩展为多个汇编行。因为在 TCP 模块具有很多的 32 位操作(序列数与应答数)，因此它在所有的协议模块中具有最大的代码尺寸。

表 2 μReTi 协议模块代码与数据尺（MSP430F149）

协议	代码（bits）	数据（bits）		数据尺寸（bits）	所占百分比
		变量	常量		
PPP	3264	616	701	4581	34%
IP	1660	110	58	1828	13%
ICMP	870	4	0	874	6%
TCP	3348	32	0	3380	25%
UDP	1450	4	1	1455	11%
API(SOCKET)	630	66	0	696	5%
Drivers &others (IF)	860	2	16	878	6%
总计	12082	834	776	13692	100%

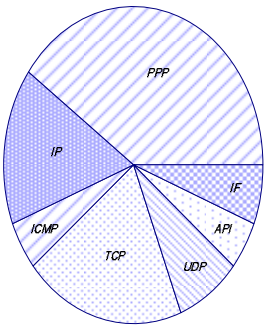

图 8 μReTi 协议模块内存占用百分比（MSP430F149）

3.2 性能比较

本节比较了 μReTi 与其它 TCP/IP 实现之间的性能，分析了它们的内存消耗以及协议功能，如表 3 所示。与标准的 TCP/IP 完全实现比较，比如 KwikNet（158KB 或 124.6KB），NicheStack（47KB）以及 NexGenIP（34.5KB），μReTi 具有最少的内存消耗；而与那些轻量的 TCP/IP 实现比较，比如 μIP（7.2KB），lwIP（13.8KB 或 22.6KB）以及 Nichlite（10KB），μReTi 具有更多的协议功能（PPP 模块）。事实上，如果移除 PPP 模块，μReTi 仅具有 6.1KB 或 9.1KW 的内存开销，μReTi 能与任何轻量的 TCP/IP 实现竞争。

在这里，我们暂时没有评估 μReTi 的 CPU 资源消耗情况，它将在后续的工作中完成。但由于放宽了不同内存空间之间的通信限制，数据复制的次数被减少，这将导致 μReTi 对 CPU 资源消耗的减少从而提高协议的性能。

表 3 不同 TCP/IP 实现的协议内存使用比较

协议（字节） / 模块	μIP[7]	lwIP[11]		Niche Lite[8]	Niche Stack[9]	KwikNet[12]		NexGen IP[13]	μReTi	
		X86	6502			X86	ARM		DSPC5410 (字)	MSP430F149
Drivers &others	1036	2281	4149	4k	16k	6k	4.6k	2.5k	614	878
ARP	1442	--	--			60k	48k	3k	--	--
UDP	--	731	1211					2k	842	1455
IP	3664	1173	1264					9k	1630	1828
ICMP		505	714						493	874
TCP		6584	11461	6k	31k			12k	2034	3380
Socket		2556	3847					6k	494	696
PPP	--	--	--	--	--	56k	40k	--	4054	4581
HTTP	1104	--	--	--	--	23k	21k	--	424	--
SNMP	--	--	--	--	--	13k	11k	--	4167	--
总计	7246	13830	22646	10k	47k	158k	124.6k	34.5k	14752	13692

4 总结与展望

本文提出 TCP/IP 精简的策略方法与设计准则，采用基于模块化的跨层设计思想，实现一套适合于资源受限的网络嵌入式设备的 TCP/IP 协议简化实现（μReTi），并根据对不同处理器的移植得出的性能评估也充分验证了 μReTi 小尺寸、低资源消耗的特性。

考虑到当前 STAR 系统的 TCP 应用特征，同时只有一个 TCP 连接以及一个可用的物理媒介，当前的 μReTi 为了减少内存需求和降低系统复杂性，并没有集成流量和拥塞控制机制。但是，如果要面向多种硬件平台的各种应用，μReTi 必须继续完善其功能以成为一个实用的 TCP/IP 协议软件。例如，在第三代 STAR 平台“Central”中，一个新的微处理器单元 CP3000 将被集成到 STAR 系统，它提供了更强的计算能力和更多的处理器内部缓存。“Central”被设计为针对各种不同的无线传感器应用。考虑到链路不同的网络通信容量以及 WSN 应用大量的节点数目，针对“Central”的 μReTi 实现必须具有 RTT 时间评测能力并支持流量和拥塞控制机制。

在另一方面，如果将嵌入式系统连接到广域网甚至 Internet，μReTi 系统的安全问题必须考虑。多级别的安全识别技术以及网络加密技术是 μReTi 系统防止网络攻击的重要方法。μReTi 协议实现的目的在于采用完全“模块化”的设计风格。不仅每一协议块能够从 μReTi 系统中轻松的增加和移除，并且协议块的每一个功能成分也应该能简单的增减。在这种方式下，μReTi 将具有更好的兼容性和系统可扩展性。针对不同的平台和应用，μReTi 实现将可以选择采用不同协议模块和不同函数功能，从而提高系统效率和通信的可靠性。

下一步我们将继续完善 μReTi 的模块化思想，最终达到可根据实际需求移除、添加相应模块的功能。同时，面对复杂的网络环境的挑战，我们将逐步实现 μReTi 的多级别安全识别技术以及网络加密技术的应用。另一方面，我们将研究 μReTi 的 IPv6 的实现，使其能够接入 IPv6 网络。

[参考文献] (References)

[1] IBM: "TCP/IP Tutorial and Technical Overview Seventh Edition"[M], International Technical Support Organization; Aug. 2001.

[2] Mauri Kuorilegto etc: Experimenting TCP/IP for Low-Power Wireless Sensor Network[R]. The 17th Annual IEEE International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC'06)， 2011.

[3] HongRiL:"Research and application of TCP/IP protocol in embedded system"[J]. Communication Software and Networks (ICCSN), 2011.

[4] H. Shrikumar: "IPic - a match head sized web-server"[EB/OL], Technical Document; URL: http://www-ccs.cs.umass. edu/~shri/iPic.html

[5] Atmel Corporation: "AVR Embedded Internet Toolkit" [EB/OL], Technical Document; URL:http://www.atmel.com, Atmel Corporation, Jan. 2011.

[6] Texas Instruments: "MSP430 internet connectivity" [EB/OL], Technical Document; Application Report, SLAA137A, URL:http://www.ti.com, Feb.2004.

[7] A. Dunkels: "uIP - A Free Small TCP/IP Implementation for 8- and 16-bit Microcontrollers" [EB/OL], Technical Document; URL: http://www.sics.se/~adam/uip.

[8] InterNiche Technologies:"NicheLite? TCP/IP(Lite) stack" [EB/OL], Technical Document; URL:http://www.iniche.com/ nichelite.php.

[9] InterNiche Technologies: "NicheStack? TCP/IP stack" [EB/OL], Technical Document; URL: http://www.iniche.com/ nichestack.php.

[10] H.Y. Zhou ,K.M. Hou, C. De Vaulx and Jian Li:"Real-time Cardiac Arrhythmias Monitoring for Pervasive Healthcare"[B]. Wireless Body Area Networks: Technology, Implementation and Applications, Pan Stanford Press, Dec. 2011.

[11] A. Dunkels: "lwIP - A Lightweight TCP/IP Stack" [EB/OL], Technical Document; URL: http://www.sics.se/~adam/lwip.

[12] KwikNet Products Ltd.: "KwikNet Reference Manual: KwikNet TCP/IP Stack User's Guide" [EB/OL], User Manual; URL: http://www.kadak.com.

[13] NexGen Software: “Portable Embedded TCP/IP Stack” [EB/OL], Technical Document; URL: http://www.nexgen-software.fr.