西北工業大學

本科毕业设计论文

题 目 面向温室群无线传感器网络路由汇聚协议的分析

专业名称 计算机科学与技术

学生姓名 裘 莹

指导教师 李 士 宁

毕业时间 2009.07

摘　要

无线传感器网络是近几年来备受关注的一种新兴的网络。这种网络功能强大，适用面广，可以为许多应用提供支持。经过近几年的研究和发展，无线传感器网络已经在精准农业控制、灾难预警与救助、环境控制和生物多样化勘测、医疗管理和卫生保健、军事侦察等领域初显成效。

无线传感器网络由多个分散的节点构成，各个节点间必须通过信息交换才能完成一定的任务，因此稳定高效的路由协议是无线传感器网络的重中之重。使用最广泛的无线传感器网络操作系统 TinyOS 自带了一种高效的路由协议实现：CTP（Collection Tree Protocol，汇聚树）协议，它用于完成无线传感器网络中最常见的数据收集任务。

本文对无线传感器网络和 TinyOS 操作系统作了简要的介绍，接着对 CTP 协议的实现进行了详细的分析，展示了它的总体架构，并对它的各个组成模块分别详加讨论，包括如下三个部分：

1. 链路估计器：用于估计一对节点间的链路质量。
2. 路由引擎：负责选择传输的下一跳。
3. 转发引擎：维护发送队列，决定发送时机，管理重传和丢包。

此外，本文还描述了使用 TOSSIM 对 CTP 协议进行仿真的过程，同时根据仿真结果验证前述的分析。文中提出了三个衡量汇聚协议性能的指标：开销、平均深度和投递率，并根据这 3 个指标比较使用不同链路估计器的汇聚协议的性能，并简要分析了产生差异的原因。

最后讲述移植 TinyOS 到自主开发节点平台的方法，并将使用 CTP 协议的数据收集程序部署到节点上投入实际应用。

关键词：　无线传感器网络，TinyOS，汇聚协议，CTP

Abstract

As a new technology, wireless sensor networks (WSN) have gained increasing attention both from the industry and research community in recent years. They are now widely used in many industrial and civilian application areas, such as precision agriculture control, disaster warning and relief, environment monitoring and biodiversity survey, health care and medical management, military reconnaissance and surveillance.

Wireless sensor network is a network consisting of many distributed sensors commumicate with each other, cooperate to archive certain goals. As a result, a robust and efficient routing protocol is crucial to WSN. TinyOS is an operating system designed for wireless sensor networks, with an efficient routing protocol implementation: CTP (Collection Tree Protocol) protocol, which is aimed to deliver small data items from every node in a network to designated collection roots.

In this paper, We analyzed the CTP protocol in detail:

1. Link Estimator: Estimate the quality between two nodes.
2. Routing Engine: Select the next hop.
3. Forwarding Engine: Maintain a queue of packets to transmit, decide when to transmit and manage retransmitting.

Additionally, we have simulated the TinyOS appllications with CTP protocol in TOSSIM and evaluated its performance according to three metrics: cost, average depth and delivery rate. Finally, we describe how to deploy this application to the npumote platform which is developed by ourselves.

Key Words: wireless sensor network, tinyos, collection protocol, CTP

目　录

第一章　绪论

1.1 研究背景

无线传感器网络（Wireless Sensor Network，WSN）是近年来新兴的一种计算机网络。这种网络由多个单节点组成，各节点通过传感或控制参数实现与环境的交互；节点必须通过相互关联才能完成一定的任务，单个节点通常无法发挥作用；节点间的关联性是通过无线通信实现的。

进入 21 世纪以来，随着无线通信、微芯片制造等技术的进步，无线传感器网络的研究也在多个方面得到了重大进展 [?]。各种技术评论杂志也一致看好 WSN 所蕴藏的巨大应用潜力和商业价值。《商业周刊》预测：WSN 和其它三项信息技术会在不远的将来掀起新的产业浪潮；非盈利性的《MIT 技术评论》将 WSN 列于十种改变未来世界新兴技术之首。美国《今日防务》杂志更认为 WSN 的应用和发展将引起一场划时代的军事技术革命和未来战争的变革。因此可以预计，WSN 的发展和广泛应用将会对人们的社会生活和产业变革带来极大的影响。我国也对传感器网络非常关注，2006 年初发布的《国家中长期科学与技术发展规划纲要》为信息技术确定了三个前沿方向，其中两个与 WSN 的研究直接相关，足见对 WSN 的重视程度。虽然 WSN 还没有得到广泛的商业使用，但已经有了不少成功应用的范例，它们从不同的侧面揭示了 WSN 的应用潜能，同时也预示了良好的商业应用前景。

1.1.1 精细农业

无线传感器网络可以应用于农业。比如将湿度和土壤组合传感器放置在农田中可以计算出精确的灌溉和施肥量。该应用所需的传感器数量比较少，大约一万平方米的面积配备一个传感器就可以了。类似的，病虫害防治得益于对农田进行高分辨率的监测。另外，WSN 也可以应用于牲畜饲养。有一个非常有趣的研究项目 [?]：在牛脖子上套上 WSN 节点，当牛接近围栏时，上面的电子装置探测到有牛接近围栏，随即模拟出驱赶牛的声音，防止牛跑出电子桩划定的放牧区域，这样放牧人便可以坐在家中轻松自在地喝咖啡看电视了。

1.1.2 结构监测

结构监测的目的是观测建筑物、轮船和飞行器等物体在外力作用下的应力响应，或者用来诊断和定位可能出现的局部损伤，是一项非常重要的工程技术。传统技术手段通过线缆将分布在物体不同部位的传感器所收集的数据汇聚到中心节点进行处理，成百上千条的线缆使监测现场异常零乱，组织试验费时费力，WSN 的出现为建筑监测提供了省时省力的技术手段。科学家利用 200

多个 Mica2 节点组成的 WSN 成功监测和评估了旧金山金门大桥的在各种自然条件下的健康状况 [?]。

1.1.3 煤矿安全监测

近年来矿井瓦斯爆炸事故频发，煤矿安全问题不容忽视。已经有研究者将无线传感网络引入煤矿安全监测和灾害预警 [?]，他们设计了一种便携式瓦斯传感器网络节点，该节点装置能够完成瓦斯浓度监测及超标报警、井下人员的实时信息采集和定位等，既可以用于工作人员查看周围环境, 也可以实现远程实时监控，从而提高了矿井作业的安全性。

1.2 研究内容

本文将使用 WSN 中应用最广泛的操作系统 TinyOS 作为研究平台。TinyOS 因具有很强的网络处理和资源收集能力而广泛应用于无线传感器网络中。数据收集是 WSN 中最常见的应用，TinyOS 中的汇聚协议可以实现传感器节点和汇聚节点之间的数据传输。感知节点采集完数据，通过该协议把数据发送给根节点。

本文的主要研究对象是 TinyOS 自带的汇聚协议——CTP (Collection Tree Protocol) 协议。通过分析 TinyOS 2.x 中 CTP 协议的源代码，论述该协议的基本原理及工作流程，期望能为无线传感器网络在农业温室群中的应用提供技术支持。本文的主要工作有如下几方面：

1. 概述 TinyOS 操作系统及 nesC 语言。提出汇聚协议的概念，分析汇聚协议要解决的问题，并从高层的视角鸟瞰 CTP 协议的总体架构。
2. 深入解析 CTP 协议中各个主要功能模块，其中包括链路估计器，路由引擎、转发引擎三个部分涉及的基本概念和工作流程，以及各部分间的相互作用。
3. 对使用 CTP 协议的应用程序进行仿真，并展示和分析仿真结果。
4. 描述如何将使用 CTP 协议的应用程序部署到自主开发的节点平台上。

1.3 章节安排

本文的章节安排如下:

第1章为绪论，主要介绍了研究背景，引出了其后的具体研究内容。

第2章概述无线传感器网络的基本概念，简单介绍了无线传感器网络操作系统 TinyOS 和 nesC 语言。随后阐明了汇聚协议的作用，展示了 TinyOS 2.x 使用的汇聚协议——CTP 协议的概貌。

第3章讲述 CTP 协议中的链路估计器部分，详细地分析了标准 LE 链路估计器的工作原理，并简单介绍了另外两种常用链路估计器的实现。

第4章详细分析了 CTP 协议中路由引擎的工作原理。

第5章详细分析了 CTP 协议中转发引擎的工作机制。

第6章介绍了使用 TOSSIM 对 CTP 协议进行仿真的方法，并提出三个指标衡量其性能。最后介绍如何将程序部署到自主开发的节点平台上。

第7章总结全文，并展望未来的研究工作。

第二章　无线传感器网络

2.1 无线传感器网络简介

2.1.1 无线传感器网络结构

无线传感器网络是由大量廉价、微型、低功耗传感器节点通过无线方式通信，自组形成的网络系统 [?]。其目的是通过节点间的协作来感知、采集和处理网络覆盖区域中感知对象的信息，并将结果发送给观察者。

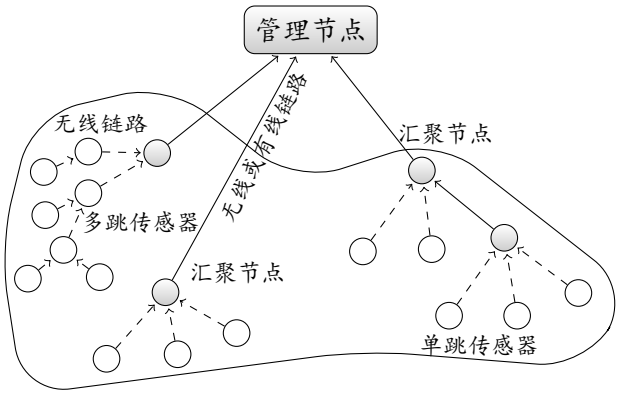


图 2.1: 典型无线传感器网络结构

常见的无线传感器网络结构如图2.1所示，传感器网络体系中通常包含传感器节点、汇聚节点和管理节点。大量传感器节点随机部署在监测区域内部或附近，能够通过自组织方式构成网络。传感器节点监测的数据沿着其他传感器节点逐跳地进行传输，在传输过程中监测数据可能被多个节点处理，经过多跳后路由到汇聚节点，最后通过互联网或卫星到达管理节点。用户通过管理节点对传感器网络进行配置和管理，发布监测任务以及收集监测数据。

2.1.2 传感器节点结构

无线传感器网络节点的体系结构如图2.2所示，它的四个基本组成部分是:传感器模块、处理器模块、无线通信模块和能量供应模块。传感器模块负责监测区域内信息的采集和数据转换；处理器模块负责控制整个传感器网络节点的操作，存储和处理本身采集的数据以及其他节点发来的数据；无线通信模块负责与其他传感器节点进行无线通信，交换控制消息和收发采集数据；能量供应模块为传感器节点提供运行所需的能量。

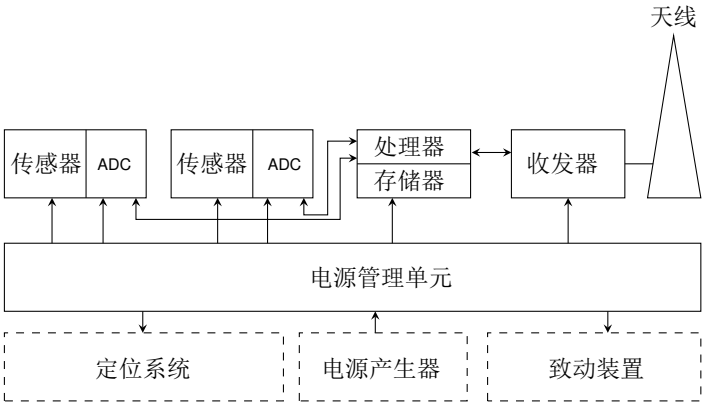


图 2.2: 传感器节点结构

2.1.3 传感器节点的限制

作为一种嵌入式设备，传感器节点在实现各种网络协议和应用时，将受到以下因素的限制：

1. 电源能力有限。传感器节点体积微小，通常携带能量十分有限的电池。而且由于节点分布广、地理环境复杂，人为更换电池补充能量是不现实的。如何高效地使用能量来最大化网络的生命周期是传感器网络面临的首要挑战。
2. 通信能力有限。传感器网络的通信带宽比较窄，通信覆盖范围只有几米到几百米，且更多地受到高山、建筑物、障碍物等地势地貌以及风雨雷电等自然环境因素的影响，通信中断频繁，容易导致通信失败。因此要求无线传感器网络的通信协议具有高效性和可靠性。
3. 计算和存储能力有限。由于体积和功耗的限制，传感器使用的嵌入式处理器和存储器一般能力有限。
4. 传感器数量大、分布范围广。传感器网络中，传感器节点密集，数量巨大，可能达到几百万、几千万，甚至更多，这使得网络的维护十分困难甚至不可维护。
5. 网络动态性强。传感器网络具有很强的动态性，网络中的传感器、感知对象和观察者这三要素都可能具有移动性，并且经常有新节点加入或已有节点失效。因此，网络的拓扑结构动态变化将导致传感器、感知对象和观察者三者之间的路径也随之变化，这就要求无线传感器网络必须具有可重构和自组织性。

2.2 TinyOS 简介

TinyOS [?] 是一款的基于组件的无线传感器网络操作系统，它是自由和开源的，目前已被学术界和工业界广泛采用，成为 WSN 研究领域事实上的标准平台。起初它是 UC Berkeley 和 Intel Research 实验室合作编写的，用来嵌入到智能微尘中。后来逐渐演变成一个国际合作项目，即 TinyOS 联盟。

TinyOS 使用 nesC [?] 语言写成。nesC（network embedded systems C）是一种基于组件、事件驱动的编程语言，用于编写 TinyOS 应用程序。它是对 C 语言的一种扩展，增加了命令（command）、事件（event）、连线（wire）、接口（interface）、配置（configuration）和模块（module）等机制 [?]。

2.2.1 TinyOS 的基本工作原理

TinyOS 的设计允许应用程序直接访问硬件。它主要解决两个问题：如何处理硬件设备中的并发数据流，如何用较小的计算和存储开销提供模块化的组件 [?]。这两个问题是很重要的，这是因为 TinyOS 要在高效支持并发操作的同时管理硬件兼容性和资源有效性。TinyOS 使用基于事件的模型用较小的存储开销支持高层的并发操作。与使用基于栈的线程方法的操作系统相比，TinyOS 具有更高的吞吐量。这是因为基于栈的线程需要为每个执行上下文保留栈空间，因此切换的速度不如 TinyOS 基于事件的方法来的快。TinyOS 可以快速创建与事件相关的任务，中间不需要阻塞或轮询。

TinyOS 包含一个小型的调度器和一系列组件。调度器用于调度组件的操作。每个组件由 4 部分组成：命令处理程序，事件处理程序，固定大小的帧和一组任务。命令和任务都在帧的上下文中执行并操纵它的状态。每个组件都会声明它的命令和事件以便与其它组件相连接。当前 TinyOS 的任务调度器使用的是简单的 FIFO 机制，但非常有效，它会在任务队列为空时让处理器进入睡眠状态，但外围设备仍处于工作状态。帧的大小是固定的并且它是静态分配的。它在编译阶段就确定了内存需求从而省去了动态分配的开销。命令是由低层组件发出的非阻塞请求，它通过返回值指示是否执行成功。

2.2.2 TinyOS 的优势

相对于其它 WSN 操作系统，TinyOS 的优势在于：

- 编写应用程序的代码量小。
- 事件传播、任务投递和上下文切换的速度快。
- 使用高效的模块化组件。

2.2.3 TinyOS 主要功能

经历了数年的发展，如今的 TinyOS 已演进到 2.1.0 版，支持数十种平台，功能和稳定性都得了不断的增强。最新版本具有的主要功能有：

1. 硬件平台抽象：TinyOS 使用三层硬件抽象以支持多平台。目前支持的平台有: eyesIFXv2，intelmote2，mica2，mica2dot，micaZ，telosb，tinynode，btnode3，iris，shimmer。
2. 任务调度：使用非抢占式的 FIFO 策略进行调度。
3. 资源虚拟化：引入通用组件和实例化组件使数据可以被重用。
4. 定时器：提供更丰富的定时器接口。
5. 通信：提供消息缓冲区结构 message_t 以及操作缓冲区的一系列接口。
6. 资源仲裁：引入 Resource 接口，可以连接到使用不同资源分配策略的仲裁器上。
7. 电源管理：实现微控制器和外围设备的电源自动管理，提供 CC2420 的低功耗监听功能。
8. 网络协议：提供两种无线传感器网络中最基本的协议——数据分发协议和汇聚协议。

2.3 汇聚协议

汇聚数据到基站是传感器网络应用程序的常见需求。常用的方法是建立至少一棵汇聚树，树根节点作为基站。当节点产生的数据要汇聚到根节点时，它只需沿着汇聚树往上发。当节点收到数据时，则将它转发给其它节点，最后一定可以到达根节点。有时汇聚协议需要根据汇聚数据的形式，检查过往的数据包，以便获取统计信息，计算聚合度和抑制重复的传输。

当网络中具有不止一个根节点时，就形成了一片森林。汇聚协议通过选择父节点隐式地使节点加入了其中一棵汇聚树中。汇聚协议一般提供了到根节点的尽力、多跳传输，它是一个任意播协议，这意味着该协议会尽力地将消息传输到任意根节点中的至少一个，但是传输并不保证必定是成功的。另外还有传到多个根节点的问题，而且数据包到达的顺序也没有保证。

由于节点的存储空间有限并且建树的算法要求是分布式的，因此协议的实现将遇到诸多挑战，主要有以下四种：

- 路由循环检测：检测节点是否选择了子孙节点作为父节点。
- 重复抑制：检测并处理网络中重复的包，避免浪费带宽。

- 链路估计：估计单跳的链路质量。
- 自干扰：防止转发的包干扰自己产生的包的发送。

TinyOS 2.x 中提供了两种汇聚协议的实现：CTP 协议和 MultiHopLQI 协议，可以较好地解决这些问题。如果需要在 TinyOS 中实现新的汇聚协议，则必须遵从 TEP（TinyOS Enhancement Proposals）119 [?] 中定义的 CollectionC 配置规范，它主要定义了汇聚协议中使用的接口和组件，将网络中节点分为四种角色：发送者，侦听者，中间处理者和接收者，CollectionC 配置的源码如下所示：

```
1  configuration CollectionC {
2      provides {
3          interface StdControl;
4          interface Send[uint8_t client];
5          interface Receive[collection_id_t id];
6          interface Receive as Snoop[collection_id_t];
7          interface Intercept[collection_id_t id];
8          interface RootControl;
9          interface Packet;
10         interface CollectionPacket;
11     }
12     uses {
13         interface CollectionId[uint8_t client];
14     }
15 }
```

2.3.1 CTP 协议

CTP（Collection Tree Protocol）协议 [?] 是 TinyOS 2.x 中自带的汇聚协议，也是实际应用中最常用的汇聚协议之一。本文将致力于 CTP 协议的分析和仿真，详细地阐述 CTP 协议的总体架构、涉及的基本概念和工作流程。

2.3.1.1 CTP 协议的实现

CTP 协议的实现主要分布在 TinyOS 2.x 安装根目录的 `tos/lib/net` 路径下，涉及的目录和文件如表2.1所示：

2.3.1.2 CTP 协议总体架构

CTP 协议可以分为三个部分：链路估计器，路由引擎和转发引擎，这三个部分的关系如图2.3所示。其中链路估计器位于最底层，负责估计两个相邻节点间的通信质量。路由引擎位于中间层，使用链路估计器提供的信息选择到根节点传输代价最小的节点作为父节点。转发引擎维护本地包和转发包的发送队列，选择适当的时机把队头的包发送给父节点。

表 2.1: CTP 协议实现文件分布

目录名	文件	作用
4bitle	LinkEstimatorP.nc	4 位链路估计器
ctp	CtpRoutingEngineP.nc CtpForwardingEngineP.nc	路由引擎 转发引擎
le	LinkEstimatorP.nc	标准链路估计器
lqi	LqiForwardingEngineP LqiRoutingEngineP	MultiHopLQI 协议

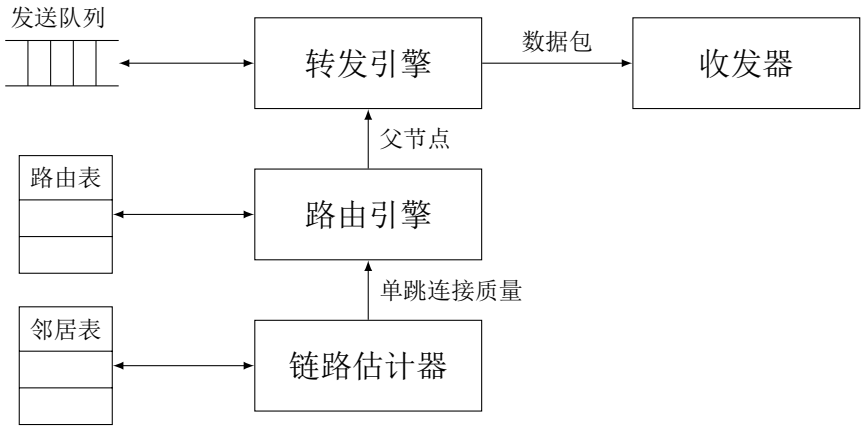


图 2.3: CTP 协议总体架构

2.3.2 MultiHopLQI 协议

MultiHopLQI 协议是一种专用于具有 CC2420 无线模块节点的汇聚协议。由于它的实现类似于 CTP 协议，本文将不专门分析 MultiHopLQI 协议，但它与 CTP 有以下几方面的不同：

1. 使用 CC2420 无线模块中的 LQI 功能作链路估计，因此它并不需要 CTP 协议中的链路估计器。
2. 没有路由表，直接使用到根节点传输代价最小的节点作为下一跳。
3. 发送队列并不遵从先进先出的原则，而是随机选择下一个发送的包。

第三章　链路估计器

3.1 简介

链路估计器主要用于估计节点间的链路质量，以供路由引擎计算路由。TinyOS 2.x 中实现的链路估计器结合了广播 LEEP 帧的收发成功率和单播数据包的发送成功率来计算单跳双向链路质量。

链路估计器的实现是多样化的，但是必须遵循 TEP 124 [?] 中的协议规范。使用网络各层中不同的信息可以得到不同的估计结果，各种方法都有其长处和缺点。在下面“实现”一节中我们将分析 3 种 TinyOS 中自带的实现。

3.2 基本概念

3.2.1 LEEP 协议

链路估计交换协议（Link Estimation Exchange Protocol，LEEP）用于在节点间交换链路估计信息，定义了交换信息使用的 LEEP 帧的详细格式。

3.2.2 入站链路质量

如图3.1(a)所示，有节点对 (A, B)，以 B 作为参考节点，A 向 B 发送的总帧数为 $total_{in}$，其中 B 成功接收到的帧数为 $success_{in}$，从而有：

$$\text{入站链路质量} = \frac{success_{in}}{total_{in}} \tag{3.1}$$

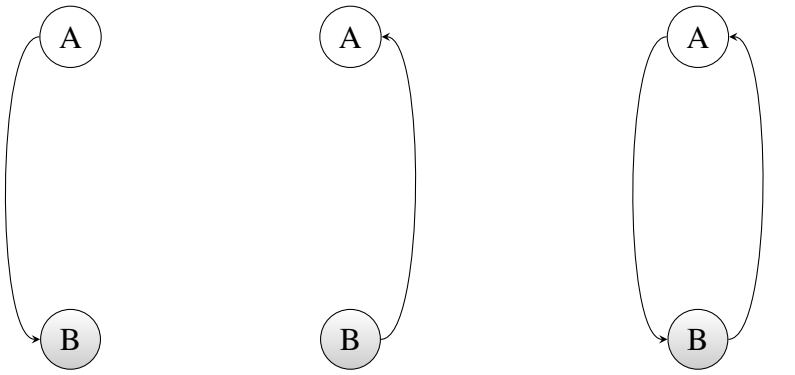
$total_{in}$ 的值可以通过 A 节点广播的 LEEP 帧中的顺序号间接计算而得。LEEP 帧中设有顺序号字段，节点 A 每广播一次 LEEP 帧，它会将该字段加 1，B 节点只需要计算连续收到的 LEEP 帧顺序号的差值就可以得到 A 总共发送的 LEEP 帧数。

入站链路质量也可以通过其它途径得到，比如 LQI 或 RSSI 之类的链路质量指示器，不过这需要无线模块支持这类功能才行。

3.2.3 出站链路质量

如图3.1(b)所示，节点对 (A, B)，以 B 作为参考点，B 向 A 发送帧数为 $total_{out}$，其中 A 成功接收到帧数为 $success_{out}$，从而有：

$$\text{出站链路质量} = \frac{success_{out}}{total_{out}} \tag{3.2}$$



(a) 入站链路质量　(b) 出站链路质量　(c) 双向链路质量

图 3.1: 三种链路质量

由于 LEEP 帧是通过广播方式发送的，节点 B 无法得知节点 A 是否收到，从而无法计算 $success_{out}$。但 B 到 A 的出站链路质量即 A 到 B 的入站链路质量，要解决该问题只有让 A 把它与 B 间的入站链路质量回馈给 B，这其实就是 LEEP 帧的主要功能之一。

TinyOS 2.x 中用 8 位无符号整数表示出站或入站链路质量。为了减少精度损失和充分利用 8 位的空间，TinyOS 2.x 在实际存储该值时对它扩大 255 倍。

3.2.4 双向链路质量

如图3.1(c)所示，对于有向节点对 (A, B)，双向链路质量定义如下：

$$\text{双向链路质量} = \text{入站链路质量} \times \text{出站链路质量} \tag{3.3}$$

本地干扰或噪声可以引起 (A, B) 和 (B, A) 链路质量不同，定义双向链路质量就是为了将这种情况考虑在内。

3.2.5 EETX 值

TinyOS 2.x 中使用 EETX（Extra Expected number of Transmission）值表示双向链路质量估计值。在 LEEP 协议中使用的有两种 EETX 值：窗口 EETX 和累积 EETX。窗口 EETX 是接收到的 LEEP 帧数或发送的数据包数达到一个固定的窗口大小时，根据窗口中的收发成功率计算出的 EETX。而累积 EETX 则是本次窗口 EETX 和上次累积 EETX 加权相加得到的。根据指数移动平均的原理让旧值的权重逐渐减少，以适应链路质量的变化，是比较符合实际的统计方法。

3.3 LEEP 协议对数据链路层的要求

LEEP 协议对数据链路层有以下三个要求：

1. 有单跳源地址
2. 提供广播地址
3. 提供 LEEP 帧长度

其中，有单跳源地址的要求是为了让收到广播 LEEP 帧的节点确定更新邻居表中哪一项的出站链路质量。现有节点的数据链路层一般都可以满足这 3 个要求。

3.4 LEEP 帧结构

根据以上分析可以得知 LEEP 帧至少应具备一个顺序号和与邻居节点间的入站链路质量。TinyOS 2.x 中实现的 LEEP 帧结构如图3.2所示：

图 3.2: LEEP 帧结构

其中 LEEP 头部结构如图3.3所示：

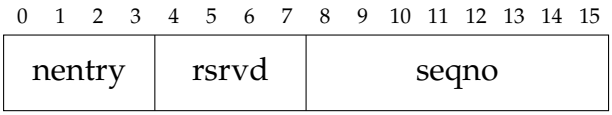


图 3.3: LEEP 帧头部

各字段定义：

- nentry：尾部的 LI 项个数
- seqno：LEEP 帧顺序号
- rsrvd：保留字段必须设为 0

链路信息项格式如图3.4所示：

各字段定义如下：

- node addr：邻居节点的链路层地址
- link quality：从与 node id 对应的节点到本节点的入站链路质量

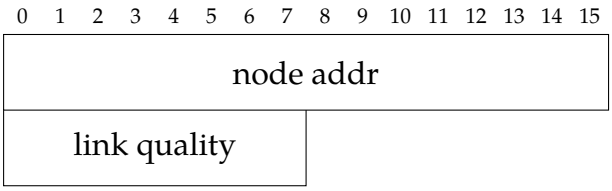


图 3.4: 链路信息项

3.5 实现

TinyOS 中可选的链路估计器有两种：标准 LE 估计器和 4 位链路估计器。可以通过更改应用程序 Makefile 中对应的路径选择使用哪一个链路估计器。

3.5.1 标准 LE 估计器

TinyOS 2.x 中标准 LE 估计器的实现在 `tos/lib/net/le` 目录下：

- `LinkEstimator.h`：头文件，包含了邻居表大小、邻居表项结构、LEEP 帧头尾结构以及 LEEP 协议中用到的常数的定义。
- `LinkEstimator.nc`：接口定义。包含了其它组件可以从 LinkEstimator 中调用的方法。由如下代码所示，这些方法可以分三类，一类用于获取链路质量，一类用于操作邻居表，还有一类用于数据包估计。

```
1  interface LinkEstimator {
2      command uint16_t getLinkQuality(uint16_t neighbor);
3      command uint16_t getReverseQuality(uint16_t neighbor);
4      command uint16_t getForwardQuality(uint16_t neighbor);
5
6      command error_t insertNeighbor(am_addr_t neighbor);
7      command error_t pinNeighbor(am_addr_t neighbor);
8      command error_t unpinNeighbor(am_addr_t neighbor);
9
10     command error_t txAck(am_addr_t neighbor);
11     command error_t txNoAck(am_addr_t neighbor);
12     command error_t clearDLQ(am_addr_t neighbor);
13
14     event void evicted(am_addr_t neighbor);
15 }
```

- `LinkEstimatorC.nc`：配置文件，用于说明链路估计器提供 LinkEstimator 接口。
- `LinkEstimatorP.nc`：LEEP 协议具体实现。

LEEP 协议的目的是为了得到本节点到邻居节点间的双向链路质量。实现中使用两种策略相结合来计算估计值，两者间的关系如图3.5所示：

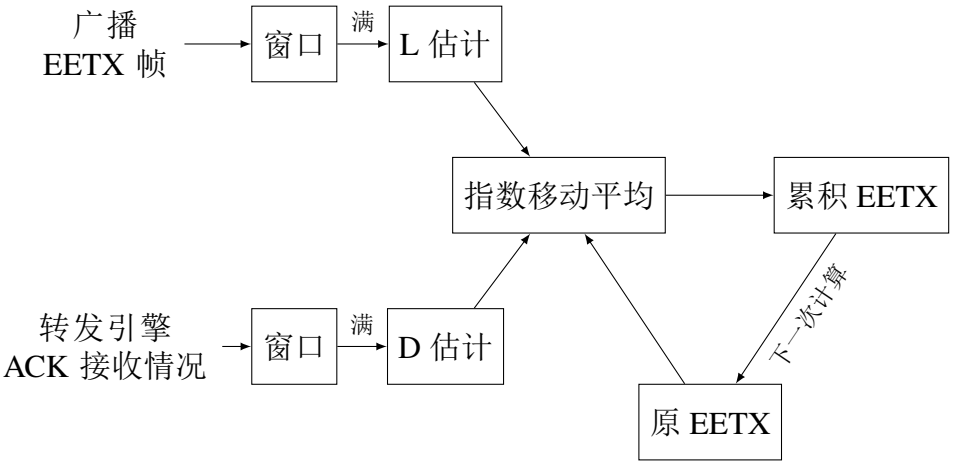


图 3.5: 链路估计值计算

3.5.1.1 根据 LEEP 帧的估计（L 估计）

LEEP 帧的估计通过 LEEP 帧的信息来估计 EETX 值。

LEEP 帧的发送：使用 Send.send() 方法，调用 addLinkEstHeaderAndFooter() 函数加 LEEP 头尾。尾部存放的是本节点到邻居节点的链路质量表，如果 LEEP 帧中一次放不下这个表，则在下次发 LEEP 帧时从首个上次放不下的表项放起，以保证每个表项有平等的发送机会。每发一个包都将帧中的顺序号字段加 1。发送的时机由 LinkEstimator 的使用者决定。

LEEP 帧的接收：每当收到一个 LEEP 帧，会触发 SubReceive.receive() 事件。处理程序根据 LEEP 头尾信息更新邻居表。这些操作集中在函数 processReceiveMessage() 中进行，该函数找到这个 LEEP 帧发送者对应的邻居表项，调用 updateNeighborEntryIdx() 函数更新收到包数计数值和丢包数计数值。其中的丢包数就是本次与上次 LEEP 帧中顺序号字段的差值。

当收到的包数达到一个固定窗口的大小时，调用 updateNeighborTableEst() 函数计算该窗口中的入站链路质量 $inquality_{win}$：

$$inquality_{win} = 255 \times \frac{\text{接收到 LEEP 帧数}}{\text{总帧数}} \tag{3.4}$$

根据动态移动平均原理更新入站链路质量：

$$inquality = \frac{\alpha \times inquality_{orig} + (10 - \alpha) \times inquality_{win}}{10} \tag{3.5}$$

TinyOS 2.x 中设衰减系数 α 值为 9，因此每次更新时，旧值占 $\frac{9}{10}$ 的权重，而新值占 $\frac{1}{10}$ 的权重。

入站链路质量发生了变化，因此需要相应地计算双向链路质量。首先计算窗口 EETX 值 $EETX_{win}$：

$$EETX_{win} = (\frac{255^2}{inquality \times outquality} - 1) \times 10 \tag{3.6}$$

为了提高存储精度，EETX 值都是扩大 10 倍存储。

接着更新累积 EETX 值：

$$EETX = \frac{\alpha \times EETX_{orig} + (10 - \alpha) \times EETX_{win}}{10} \tag{3.7}$$

3.5.1.2 根据数据包的估计（D 估计）

根据数据包的估计通过发送数据包的成功率来估计 EETX 值。

LinkEstimator 并不能得知上层的数据包是否发送成功。因此它提供两个命令 txAck() 和 txNoAck() 让上层组件调用。txAck() 用于告知链路估计器数据包发送成功，它将对应通信邻居的成功传输数据包计数值和总传输包计数值加 1。当总传输包数达到一个固定窗口的大小时，调用 updateDEETX() 函数计算窗口 EETX 值 $DEETX_{win}$：

$$DEETX_{win} = (\frac{\text{总包数}}{\text{成功包数}} - 1) \times 10 \tag{3.8}$$

接着更新累积 EETX 值：

$$EETX = \frac{\alpha \times EETX_{orig} + (10 - \alpha) \times DEETX_{win}}{10} \tag{3.9}$$

3.5.2 四位链路估计（4BITLE）

四位链路估计 [?] 与标准 LE 估计器的实现在结构上大体上是相同的。有所改进的是它提炼了物理层、链路层和网络层回馈的信息用于更精确的链路估计。其中 1 位来自物理层，1 位来自链路层，2 位来自网络层。4BITLE 的实现在 `tos/lib/net/4bitle` 目录下。

3.5.2.1 物理层

在物理层，我们可以从一个包的传输中测量信道的质量。通常情况下，接收到错误位比较少的包很可能比错误多的包链路质量好。物理层的测量是快速并且廉价的，可以避免估计器在网络边缘或劣质的连接上浪费精力。我们可以把它提炼为一个 white 位，它表示在接收包时信道链路质量的优劣程度。

3.5.2.2 链路层

在链路层，我们可以测量包是否被传输并确认。基于广播探测的估计器面临着一个问题，它们将链路估计和数据通信分开实现，如果链路质量变坏导致包丢失，链路估计器要直到下一个路由信标被丢弃时才能反映这种变化。为了解决该问题，我们使用一个 ack 位，表示节点在一次传输中是否收到链路层的确认。

3.5.2.3 网络层

在网络层，我们可以了解到哪个连接是高层性能上最有价值的。如果没有网络层的信息，估计器可能选择一条路由环路，或者在最坏的情况下会与网络失去连接。无线传感器网络很可能会因链路层和网络层链路表的不一致而导致故障。我们可把这些提炼为 2 个位：

- pin 位： 用于告诉估计器不能剔除正在使用的连接。
- compare 位： 用于告诉网络层这个连接看起来颇有希望。

3.5.3 链接质量指示（LQI）

链接质量指示（Link Quality Indicator, LQI）是 MultiHopLQI 汇聚协议中用于链路估计的部分。它需要无线模块支持 LQI，因此它只适用于 CC2420 之类具有链接质量指示器的节点。它完全使用物理层的信息作链路估计。

第四章　路由引擎

4.1 简介

路由引擎的责任是选择传输的下一跳。一个理想的路由引擎应当可以选择到根节点跳数尽量少而连接质量尽量好的传输路径，这样可以减少转发次数和丢包率，从而降低传感器网络的能量消耗，延长网络的生存期。但由于节点的的存储容量和处理能力一般都非常有限，难以存储大量的路由信息和使用复杂的路由算法，故而有线网络中常用的路由算法如 TCP/IP 中的 OSPF 和 RIP 协议 [?] 在这里是不适用的。传感器网络的路由设计注重简单有效，使用有限的资源达到最好的效果。TinyOS 2.x 中 CTP 协议实现的路由引擎可以较好地实现这个目标。它用于建立到根节点的汇聚树，利用链路质量估计器提供的信息合理地选择下一跳节点，使采样节点到根节点的传输次数尽可能地少。

4.2 基本概念

4.2.1 路径 ETX

路径 ETX(Expected number of Transmissions) 是父节点到根节点的 ETX 与本节点与父节点间的单跳 ETX 之和，如图4.1所示。单跳 ETX 与链路估计器提供的 EETX 值关系为 ETX=EETX+1。它的大小可以反映出到根节点的跳数。在一般情况下，路径 ETX 越小说明离根节点越近，路由引擎正是根据这一事实选择 ETX 最小的邻居作为父节点，以期获得到根节点的最少传输次数。

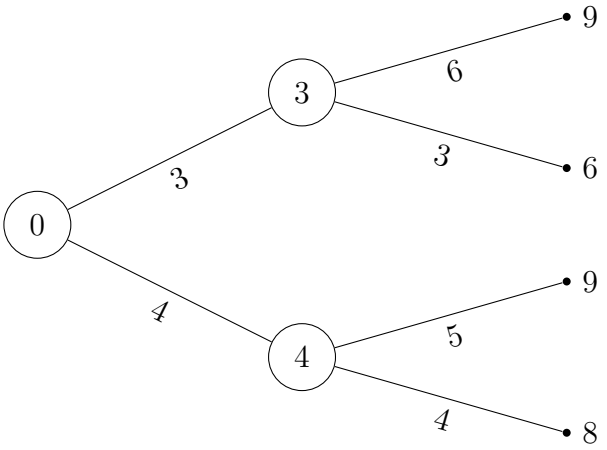


图 4.1: 路径 ETX 值计算

邻居节点地址	路由信息		
	父节点地址	ETX	拥塞
0			
1			
2	…	…	…

图 4.2: 路由表结构

4.2.2 路由表

路由表是路由引擎的核心数据结构。CTP 协议中使用的路由表结构如图4.2所示的形式，它存储了邻居节点信息，主要是邻居的路径 ETX 值。路由表的大小取决于链路估计器邻居表的大小，因为不在邻居表的节点无法作为邻居节点进入路由表。

4.2.3 CTP 路由帧（信标帧）

路由引擎用广播的形式发送路由帧以便在节点间交换路由信息。

CTP 路由帧格式如图4.3所示：

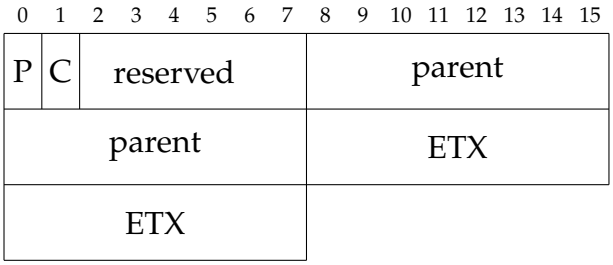


图 4.3: CTP 路由帧格式

各字段意义如下：

- P：取路由位。如果节点收到一个 P 位置位的包，它应当尽快传输一个路由帧。
- C：拥塞标识。如果节点丢弃了一个 CTP 数据帧，则必须将下一个传输路由帧的 C 位置位。
- parent：节点的当前父节点
- ETX：节点的当前 ETX 值

当节点接收到一个路由帧时，它必须更新路由表相应地址的 ETX 值。如果节点的 ETX 值变动很大，那么 CTP 必须传输一个广播帧以通知其它节点更新它们的路由。与 CTP 数据帧相比，路由帧用父节点地址代替了源节点地址。父

节点可能发现子节点的 ETX 值远低于自己的 ETX 值的情况，这时它需要准备尽快传输一个路由帧。

4.2.4 当前路由信息

记录当前使用的父节点的信息。如它的 AM 层地址，路径 ETX 等。

4.3 实现

TinyOS 中路由引擎的实现在 `tos/lib/net/ctp` 目录的下列文件中：

- `CtpRoutingEngineP.nc`：路由引擎的具体实现
- `TreeLouting.h`：路由引擎中使用的一些结构和常数的定义
- `Ctp.h`：路由帧结构的定义

4.3.1 使用的接口和提供的组件

从下列源码中可以看到，CtpRoutingEngineP 是一个通用组件，可以通过参数设定路由表大小、信标帧发送的最小和最大间隔。它使用了链路估计器、两个定时器和一些包收发处理接口；提供的接口主要是 Routing 路由接口，它包含了一个最重要的命令 nexthop() 用于为上层组件提供下一跳的信息。

```
1  generic module CtpRoutingEngineP(uint8_t routingTableSize,
2                  uint32_t minInterval, uint32_t maxInterval) {
3      provides {
4          interface UnicastNameFreeRouting as Routing;
5          interface RootControl;
6          interface CtpInfo;
7          interface StdControl;
8          interface CtpRoutingPacket;
9          interface Init;
10     }
11     uses {
12         interface AMSend as BeaconSend;
13         interface Receive as BeaconReceive;
14         interface LinkEstimator;
15         interface AMPacket;
16         interface SplitControl as RadioControl;
17         interface Timer<TMilli> as BeaconTimer;
18         interface Timer<TMilli> as RouteTimer;
19         interface Random;
20         interface CollectionDebug;
21         interface CtpCongestion;
22
23         interface CompareBit;
24     }
25 }
```

4.3.2 信标帧定时器与路由定时器

4.3.2.1 信标帧定时器

信标帧定时器（BeaconTimer）用于周期性的发送信标帧。发送间隔是指数级增长的。初始的间隔是一个常数 minInterval（其值为 128），在每更新一次路由信息后，将间隔加倍。因此随着网络的逐渐稳定，将很少看到节点广播信标帧。定时器间隔在使用指数级增长的基础上还加上随机数以错开发送信标帧的时机，避免节点同时发送信标帧导致信道冲突。此外，定时器可以重置为初始值，这主要用于处理一些特殊情况，比如节点收到一个 P 位置位的包要求尽快发信标帧，或者提供给上层使用者重置间隔的功能。

4.3.2.2 路由定时器

路由定时器（RouteTimer）用于周期性的启动更新路由任务。更新间隔固定为一个常数 BEACON_INTERVAL，其值为 8192。该定时器触发后将启动更新路由选择任务。

4.3.3 发送信标帧与更新路由选择任务

4.3.3.1 发送信标帧任务

由信标帧定时器触发。以广播的方式告知其它节点本节点的 ETX 值、当前父节点和拥塞信息。

4.3.3.2 更新路由选择任务

更新路由选择任务一般由路由定时器触发，但也可以在其它条件下触发，如信标帧定时器到期、重新计算路由、剔除了某个邻居等需要更新路由选择的情况下触发。更新路由选择任务通过遍历路由表找出路径 ETX 值最小的节点作为父节点，并且该节点不能是拥塞的或是本节点的父节点。

4.3.4 信标帧接收事件

BeaconReceive.receive() 事件，在收到其它节点的信标帧时触发。它将根据信标帧的发送者和 ETX 值更新相应的路由表项。如果收到的是根节点的信标帧，则调用链路估计器将它固定在连接表中。如果信标帧的 P 位置位，则重设信标帧定时器，以便尽快广播本节点的信标帧让请求者收到。

4.3.5 工作流程分析

路由引擎工作流程如图4.4所示。节点启动时将初始化路由引擎。路由引擎通过将 Init 接口接到 MainC 的 SoftwareInit 接口以实现节点启动时自动初始化路由引擎。初始化的工作有：初始化当前路由信息、初始化路由表为空，初始化路由帧消息缓冲区以及一些状态变量等。

应用程序通过 StdControl 接口的 start() 方法正式启动路由引擎，这将启动两个定时器 RouteTimer 和 BeaconTimer。其中 RouteTimer 的时间间隔设为 BEACON_INTERVAL（8192），BeaconTimer 的下一次发送时间初始值设为 minInterval（128）。

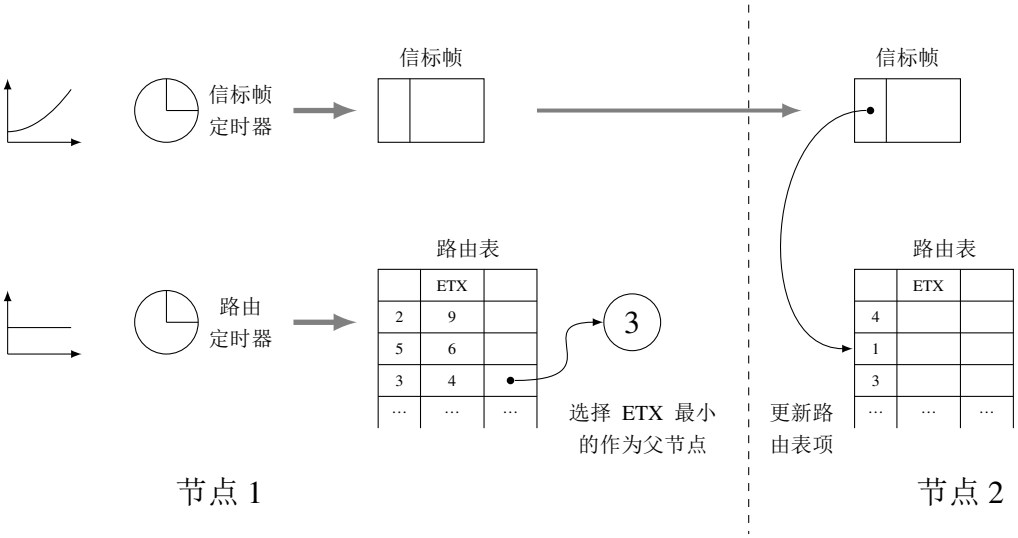


图 4.4: 路由引擎工作流程

由于 BeaconTimer 的触发时间值设置的比 RouteTimer 的触发间隔小的多，因此 BeaconTimer 将率先触发，并投递 updateRouteTask() 以更新路由选择，接着投递 sendBeaconTask() 任务发送信标帧。此后，RouteTimer 以恒定的时间间隔触发并投递 updateRouteTask()，而 BeaconTimer 触发后会将下次触发的时间间隔加倍。

除了定时器在不断的触发以投递任务外，路由引擎还需要处理其它节点的信标帧。当接收到一个广播的信标帧时，会触发 BeaconReceive.receive() 事件并根据信标帧中的发送者和它的 ETX 值更新相应的路由表项。

另外，如果链路估计器剔除了一个侯选邻居，则路由引擎也要相应地从路由表把该邻居移除，并更新路由选择，从而保证了路由表和连接表的一致性。

第五章　转发引擎

5.1 简介

转发引擎主要负责以下 5 种工作：

1. 向下一跳传递包，在需要时重传，同时根据是否收到 ACK 向链路估计器传递相应信息。
2. 决定何时向下一跳传递包
3. 检测路由中的不一致性，并通知路由引擎
4. 维护需要传输的包队列，它混杂了本地产生的包和需要转发的包。
5. 检测由于丢失 ACK 引起的单跳重复传输

5.2 基本概念

5.2.1 路由循环

路由循环是指某个节点将数据包转发给下一跳，而下一跳节点是它的子孙节点或者它本身，从而造成了数据包在该环路中不断循环传递，如图5.1所示，由于节点 E 在某个时刻错误地选择了 H 节点作为父节点，从而造成了路由循环。

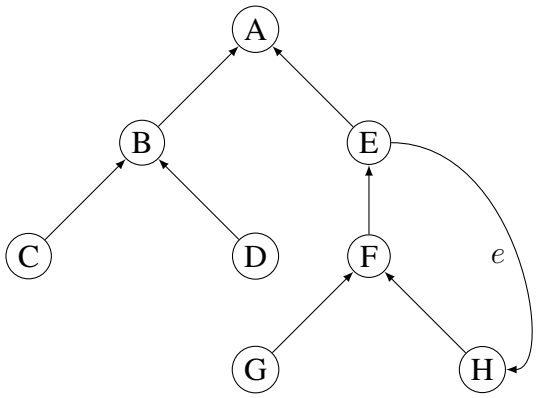


图 5.1: 路由循环

5.2.2 包重复

包重复是指节点多次收到具有相同内容的包。这主要是由于包重传引起的。比如发送者发送了一个数据包，接收者成功地收到了该数据包并回复 ACK，但 ACK 在中途丢失，因此发送者会将该包再一次发送，从而在接收者处造成了包重复现象。

5.2.3 CTP 数据帧

CTP 数据帧是转发引擎在发送本地数据包时所使用的格式。它在数据包头增加一些字段用于抑制包重复和路由循环。

CTP 数据帧格式如图5.2所示：

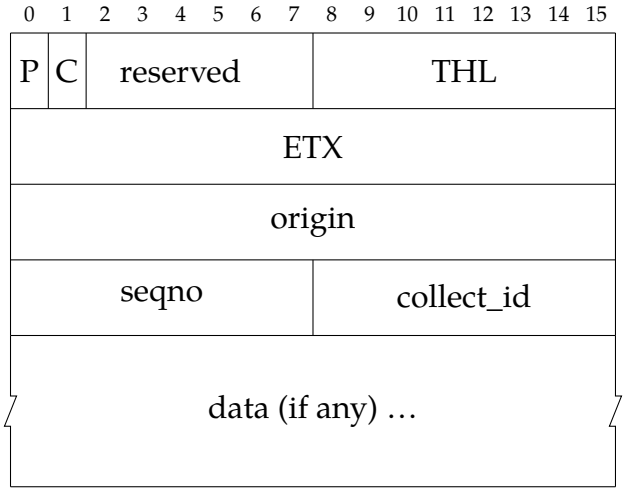


图 5.2: CTP 数据帧格式

各字段定义如下：

- P：取路由位。P 位允许节点从其它节点请求路由信息。如果节点收到一个 P 位置位的包，它应当传输一个路由帧。
- C：拥塞标志位。如果节点丢弃了一个 CTP 数据帧，它必须在下一个传输的数据帧中置 C 位。
- THL：已存活时间（Time Have Lived），它主要用于解决路由循环问题。当节点产生一个 CTP 数据帧时，它必须设 THL 为 0。当节点接收到一个 CTP 数据帧时，它必须增加 THL 值。如果节点接收到的数据包 THL 为 255，则将它回绕为 0。该字段主要用于解决数据包在环路中停留太久的问题，但在当前版本的 CTP 协议中暂时还没有实现这一功能。
- ETX：单跳发送者的 ETX 值。当节点发送一个 CTP 数据帧时，它必须将到单跳目的地的路由 ETX 值填入 ETX 字段。如果节点接收到的路由梯度比自己的小，则它必须准备发送一个路由帧。
- origin：包的源地址。转发的节点不可修改这个字段。
- seqno：源顺序号。源节点设置了这个字段，转发节点不可修改它。
- collect_id：高层协议标识。源节点设置了这个字段，转发节点不可修改它。

• data：数据负载。0 个或多个字节。转发节点不可修改这个字段。

origin, seqno, collect_id 合起来标识了一个唯一个源数据包，而 origin, seqno, collect_id, THL 合起来标识了网络中唯一一个数据包实例。两者的区别在路由循环中的重复抑制是很重要的。如果节点抑制源数据包，则它可能丢弃路由循环中的包；如果它抑制包实例，则它允许转发处于短暂的路由循环中的包，除非 THL 凑巧回绕到与上次转发时相同的状况。

5.2.4 队列项

队列项（Queue Entry）中存放了对应消息的指针、对应的发送者和可重传次数。本地包与转发包的队列项分配方法有所不同：转发包的队列项是通过缓冲池分配的，而本地包的队列项是编译期间静态分配的。

5.2.5 消息发送队列

消息发送队列结构是转发引擎的核心结构。它存放了队列项的指针，队头元素指向的队列项中的消息将被优先发送。

5.2.6 缓冲池

缓冲池是操作系统中用于统一管理缓冲区分配的一个设施 [?]。应用程序可以使用缓冲池提供的接口方便地获取和释放缓冲区。对于不能动态分配存储空间的 TinyOS 来说，这一点非常有价值，因为它可以重复利用一段静态存储空间。

转发引擎中使用了两个缓冲池：队列项缓冲池（QEntryPool）和消息缓冲池（MessagePool）。队列项缓冲池用于为队列项分配空间，如图5.3所示，当转发引擎收到一个需要转发的消息时，它会从队列项缓冲区中取出一个空闲的队列项，作相应的初始之后把队列项的指针放入消息队列队尾。在成功地发送了一个消息并收到 ACK 或消息重发次数过多被丢弃时，转发引擎会从队列项缓冲池中释放这个消息对应的队列项，使它变为空闲，因此队列缓冲池就可以把这块空间分配给后续的消息。

消息缓冲池的工作原理与队列项缓冲池类似，只不过它存的是消息结构。消息缓冲池有一个缓冲区交换的行为，在“缓冲区交换”一节中详细论述。

在 TinyOS 2.x 中，消息缓冲池的初始大小设定为一个常数 FORWARD_COUNT（值为 12）。队列项缓冲池的初始大小为 CLIENT_COUNT + FORWARD_COUNT，其中 CLIENT_COUNT 是 CollectionSenderC 使用者的个数，加上它是考虑到了本地产生的包也会进入发送队列，而本地包的最大个数正是 CollectionSenderC 使用者的个数，这样就保证了发送队列不会因为本地发送者太多而不断产生溢出。如果不考虑这个因素，则在本地发送者很多的情况下节点可能产生拥塞的假象。

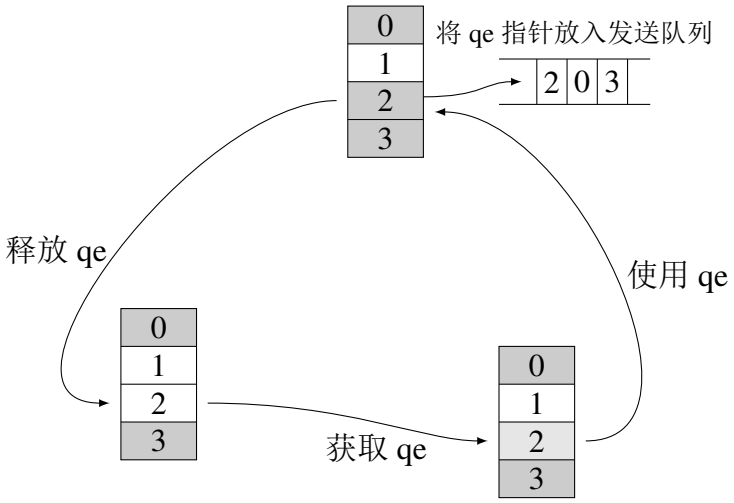


图 5.3: 从缓冲池分配和释放队列项

5.2.7 缓冲区交换

缓冲区交换是转发过程中一个比较微妙的环节。如图5.4所示，从缓冲池中获得的消息结构并不是直接用于存储当前接收到的消息，而是用于存储下一次收到的消息。由于当前接收到的消息必定已经有了它自己的存储空间，因此只要让相应的队列项指向它就可以找到这个消息的实体。但是下一个接收到的消息就不应该存储在这一块空间，而缓冲区交换正是用于为下一次收到的消息分配另外一块空闲的存储空间。传统的做法通常是设置一个消息结构用于接收消息，每当收到一个消息后将它整个复制到空闲存储空间中。相比之下，缓冲区交换可以省去一次复制的开销。

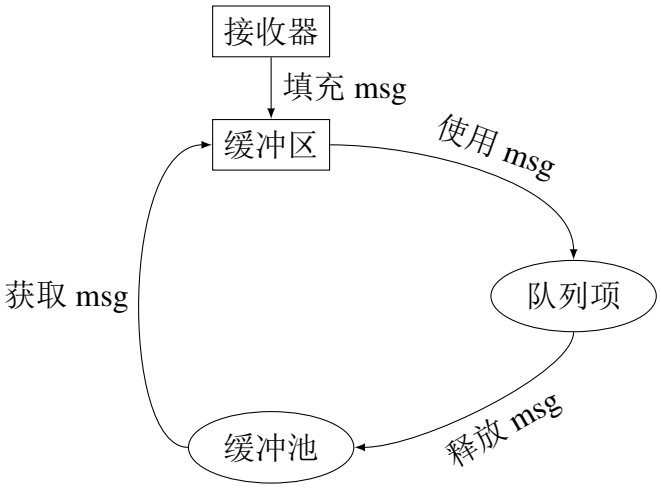


图 5.4: 缓冲区交换

5.3 实现

组件 tos/lib/net/ctp/CtpForwardingEngineP.nc 实现了转发引擎。

5.3.1 使用的接口和提供的组件

从下列源码中可以看到，CtpForwardingEngine 使用了路由引擎提供的接口 UnicastNameFreeRouting 用于得到下一跳信息，使用了系统提供的 Queue、Pool、SendCache 接口分别实现消息发送队列、队列项缓冲池、消息缓冲池和发送消息缓存，同时也使用了 LinkEstimator 用于向链路估计器反馈数据包发送成功与否的信息。

```
1  generic module CtpForwardingEngineP() {
2      provides {
3          interface Init;
4          interface StdControl;
5          interface Send[uint8_t client];
6          interface Receive[collection_id_t id];
7          interface Receive as Snoop[collection_id_t id];
8          interface Intercept[collection_id_t id];
9          interface Packet;
10         interface CollectionPacket;
11         interface CtpPacket;
12         interface CtpCongestion;
13     }
14     uses {
15         interface AMSend as SubSend;
16         interface Receive as SubReceive;
17         interface Receive as SubSnoop;
18         interface Packet as SubPacket;
19         interface UnicastNameFreeRouting;
20         interface SplitControl as RadioControl;
21         interface Queue<fe_queue_entry_t*> as SendQueue;
22         interface Pool<fe_queue_entry_t> as QEntryPool;
23         interface Pool<message_t> as MessagePool;
24         interface Timer<TMilli> as RetxmitTimer;
25         interface LinkEstimator;
26         interface Timer<TMilli> as CongestionTimer;
27         interface Cache<message_t*> as SentCache;
28         interface CtpInfo;
29         interface PacketAcknowledgements;
30         interface Random;
31         interface RootControl;
32         interface CollectionId[uint8_t client];
33         interface AMPacket;
34         interface CollectionDebug;
35         interface Leds;
36     }
37 }
```

CtpForwardingEngine 提供的接口分别为网络中的四种扮演不同角色的节点服务。为发送者提供 Send 接口，为侦听者提供 Snoop 接口，为网络处理者提供 Intercept 接口，为接收者提供 Receive 接口。

5.3.2 关键函数

转发引擎的四个关键函数为包接收 SubReceive.receive()，包转发 forward()，包传输 SendTask() 和包传完之后的善后工作 SubSend.sendDone()。

5.3.2.1 receive() 函数

receive() 函数决定节点是否转发一个包。它有一个缓冲区缓存了最近收到的包，通过检查这个缓冲区可以确定它是否是重复的。如果不是，则调用 forward() 函数进行转发。

5.3.2.2 forward() 函数

forward() 函数格式化需要转发的包。它检查收到的包是否有路由循环，使用的方法是判断包头中的 ETX 值是否比本节点的路径 ETX 小。接着检查发送队列中是否有足够的空间，如果没有，则丢弃该包并置 C 位。如果传输队列为空，则投递 SendTask 任务准备发送。

5.3.2.3 SendTask 任务

SendTask 任务检查位于发送队列队头的包，请求路由引擎的路由信息，为到下一跳的传输作好准备，并将消息提交到 AM 层。

5.3.2.4 sendDone 事件

当发送结束时，sendDone 事件处理程序会检查发送的结果。如果包被确认，则将包从传输队列中取出。如果包是本地产生的，则将 sendDone 信号向上传。如果包是转发的，则将该消息结构释放到消息缓冲池。如果队列中还有剩余的包（比如没有被确认的），它启动一个随机定时器以重新投递这个任务。该定时器实质上用于限制 CTP 的传输速率，不让它尽快地发包，这是为了防止在通路上自我冲突。

5.3.3 工作流程分析

当转发引擎收到一个转发包时，它会检查该数据包是否在缓存或发送队列中，这主要是为了抑制包重复。如果不是重复包，则调用 forward() 进行转发。forward() 函数为该消息在消息池中分配队列项和消息结构，然后把队列项指针放入发送队列。如果此时投递发送消息任务的定时器没有运行，则立即投递发送消息任务，以选取发送队列队头的数据包进行发送。发送成功之后将触发 sendDone 事件做一些善后工作，比如检查刚发送的包是否收到链路层 ACK，如果收到，则从队列中删除这个包的队列项，并释放相关资源。如果没有收到

ACK，则启动重传定时器，再一次投递发送消息任务进行重传。若重传次数超过 CLIENT_COUNT 次，则丢弃该包。

5.3.4 本地数据包的发送

转发引擎也负责本地数据包的发送。应用程序通过使用 CollectionSenderC 组件发送本地包。nesC 编译器会根据 CollectionSenderC 组件使用者的个数为每个使用者静态地分配一个队列项，并用一个指针数组指向各自的队列项。如果某个使用者需要发送数据包，则先检查它对应的指针是否为空。若为空，则说明该使用者发送的前一个数据尚未处理完毕，返回发送失败；若不为空，则说明它指向的队列项可用，用数据包的内容填充队列项并把它放入发送队列等待发送。

第六章　仿真与部署

6.1 仿真

6.1.1 TOSSIM 简介

TOSSIM [?] 是一个 TinyOS 程序仿真工具。它是一个库，你可以写程序调用并运行以实现仿真。TOSSIM 支持两种编程接口：Python 和 C++。Python 可以交互式动态地仿真，如同一个强力的调试器。如果对仿真的时间性能要求较高，则可以用 C++ 接口。

TOSSIM 的工作原理是通过替换系统组件实现仿真，具体替换哪个组件视情况而定。比如定时器的仿真可以使用替换 HilTimerMillic 组件的方法实现，也可以通过替换 atmega128 平台硬件时钟的 HPL(Hardware Presentation Layer) 实现。前者是对任何平台通用的，但是缺乏逼真度。后者通过仿真芯片的行为实现，逼真度高，但是只对 atmega128 适用。TOSSIM 是一个离散事件仿真器，它从事件队列（以发生时间排序）中取出事件并运行之。仿真事件可以是底层硬件中断或高层的系统事件（如包接收事件），也可以是任务。

6.1.2 仿真 CTP 协议

我们使用 TinyOS 2.x 中自带的示例 `apps/tests/TestNetwork` 作仿真。该程序使用 CTP 协议将节点收集到的数据通过汇聚树汇聚到任意一个根节点。我们使用 Python 编写测试脚本。

6.1.2.1 让节点间可以通信

不做任何设置的话，TOSSIM 中的节点是无法互相通信的。因此我们要先配置网络拓扑结构。TOSSIM 默认使用基于信号强度的模型，需要有每两个节点间的增益值，这可以用 radio 对象中的 add() 方法实现的，代码如下：

```
1       t = Tossim([])
2       r = t.radio()
3       r.add(src, dest, gain)
```

其中 src 是源节点，dest 是目的节点，gain 是源到目的的增益。由于源到目的与目的到源的增益可能是不同的，因而要分开指定。

一般路由协议的仿真，网络中都会有至少上百个节点，手动一个个添加增益是不大现实的。我们用一个文件记录所有节点对间的增益，一行一个，每行的格式如下：

```
gain <源节点号> <目的节点号> <增益>
```

用 python 脚本可以轻松地将这些增益值添加到 radio 对象，假设文件名为 topo.txt，源代码如下所示：

```
f = open("topo.txt","r")
lines = f.readlines()
for line in lines:
    s = line.split()
    if (len(s) > 0):
        if s[0] == "gain":
            r.add(int(s[1]), int(s[2]), float(s[3])
```

另外，TOSSIM 使用 CPM 算法 [?] 仿真 RF 模块的噪声。该算法需要先读入若干个噪声记录，然后生成噪声模型。我们使用斯坦福大学 Meyer 实验室提供的噪声记录 meyer-heavy.txt，它是每行一个噪声值。接下来我们先为 10 个节点添加各自的噪声值：

```
noise = open("meyer-heavy.txt", "r")
lines = noise.readlines()
for line in lines:
    str = line.strip()
    if (str != ""):
        val = int(str)
        for i in range(0, 10):
            m = t.getNode(i)
            m.addNoiseTraceReading(val)
```

再用 CPM 算法为每个节点生成噪声模型：

```
for i in range(0,10):
    t.getNode(i).createNoiseModel()
```

现在节点间终于可以通信了。

6.1.2.2 生成仿真结果

保证节点可以通信之后就可以对 CTP 协议进行仿真。进入 `apps/test-s/TestNetwork` 目录，运行 `make micaz sim` 生成仿真库。编写测试脚本 test.py，在 shell 中使用以下命令运行：

```
export PYTHONPATH=$TOSROOT/support/sdk/python
python test.py
```

就可以就看到路由引擎和链路估计的调试信息。

6.1.2.3 可视化仿真

上述小节已经搭建好了 CTP 协议仿真的环境，可以得到详细的调试信息。但是调试信息只是一些文本信息，很不直观，难以从中领会 CTP 的工作流程。因此我们使用 3D 动画生成软件 Ubigraph 对仿真结果进行可视化演示。

Ubigraph[?] 可以编程控制各种立体结构，并且自动布局，是一个理想的 3D 演示工具。它有 linux 和 mac 版，使用 xmlrpclib 实现，对 python 语言的支持最完整。

CTP 仿真可视化程序基本思想是每一条调试信息对应一个演示动作。比如转发引擎 Forwarder 转发一个包时会有如下调试信息：

```
DEBUG (3): CtpForwardingEngineP$0$SubSend$sendDone to 2 and 0
```

其中 DEBUG(3) 表示是 TOS_ID 为 3 的节点所产生的调试信息，后面 2 和 0 的意义对照源码可以知道 2 是转发的目标，0 表示转发成功。可以用正则表达式来匹配并提取有用的信息，然后执行相应的演示动作。

6.1.3 仿真结果

下面通过仿真给出 CTP 协议生成的网络拓扑结构，并分析开销、汇聚树的平均深度和包投递率。

6.1.3.1 拓扑结构

为了节省篇幅，我们选取了 10 个节点作仿真。表6.1列出了 10 个节点间的双向增益。由于信号必定是会衰减的，因此增益值都是负值（除了理想状况下没有衰减，增益值为 0），增益值越小说明链路质量越差，当增益值小于 -85dBm 时，节点间几乎无法通信。节点与自身之间的增益统一设为 0，这与节点必定能收到它发送给自己的包的事实是相符的。

表 6.1: 节点间连接质量 (dBm)

节点号	0	1	2	3	4	5	6	7	8	9
0	0	-70	-83	-83	-96	-98	-98	-100	-102	-110
1	-71	0	-77	-89	-98	-94	-102	-104	-103	-108
2	-84	-77	0	-76	-91	-95	-91	-99	-101	-105
3	-78	-83	-70	0	-74	-78	-85	-92	-95	-95
4	-93	-93	-87	-75	0	-77	-83	-84	-89	-100
5	-94	-90	-90	-80	-77	0	-71	-86	-91	-93
6	-98	-100	-90	-89	-86	-74	0	-74	-84	-92
7	-99	-101	-96	-95	-86	-88	-73	0	-77	-84
8	-99	-99	-97	-96	-90	-92	-82	-76	0	-76
9	-109	-106	-103	-99	-102	-96	-92	-84	-78	0

表6.2展示了具有网络中汇聚树建立的过程，这体现在父节点选择的变化过程上。其中根节点节点号为 0，‘?’表示还没有找到父节点。

首先，与根节点通信质量最好的节点 1 和节点 2 首先与根节点建立起了路由。接着节点 2 与通信质量最好的节点 3 建立了连接，随后其它节点通过 3、7 等节点的转发也建立起了到根的路由。在这个过程中，有些节点不只一次地更

换了父节点，比如节点 5 经常性地在节点 3、4 间切换，这主要是由于 5 与 3、4 之间的通信质量十分相似，而这两条信道中都存在程度相近的噪声干扰。

表 6.2: 汇聚树建立过程

节点号	原父节点	新父节点	节点号	原父节点	新父节点
1	?	0	5	3	4
2	?	0	7	6	3
3	?	2	5	4	3
4	?	3	5	3	4
5	?	3	8	7	5
6	?	5	5	4	3
5	3	4	5	3	4
5	4	3	8	5	7
7	?	6	6	5	7
5	3	4	5	4	3
5	4	3	5	3	4
5	3	4	2	0	1
3	2	0	3	0	2
8	?	7	5	4	6
9	?	8	2	1	0
5	4	3			

最终，网络中的节点形成的一棵相对稳定的汇聚树，任意一个节点都存在到根节点的一条通路。汇聚树的拓扑结构如图6.1所示。

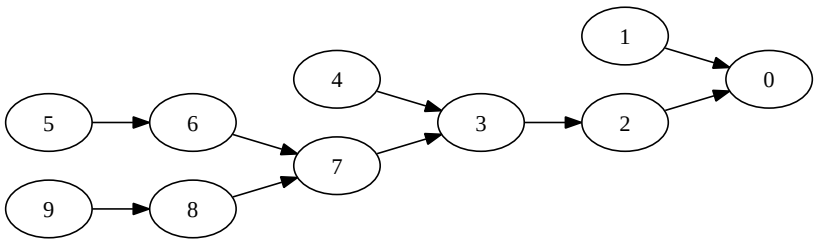


图 6.1: 汇聚树节点拓扑结构

6.1.3.2 CTP 协议性能评价

评价 CTP 协议的性能主要根据 3 个指标：开销，平均深度和包投递率。

开销是指平均每个单播包传输的总次数。由于包传输对节点来说所消耗的能量是相当可观的，因此开销的大小将直接影响到网络的生存时间，开销越大，网络的生存时间越短。它与路径的跳数，每个连接的重传数以及由于包丢失而造成的浪费有关。

平均深度指的是汇聚树的平均深度。如果所有的连接是完好的，没有重传和丢包，那么平均深度将是开销的下界。两者间的差异意味着选择连接的质量，这可能是由于重传或丢包所造成的，高效的路由算法可以使这个差异最小化。

投递率是指根节点收到的不重复消息的百分率，即根节点收到的数据包中除去重复的包所剩余的包数与所有节点发出的本地数据包数之比。这个指标的大小可以体现出丢包数，投递率越高说明丢包数越少。

在仿真结果中，可以通过监视节点转发引擎中的转发事件（包括了本地发送的包），计算出本地包的个数和转发次数。平均开销计算公式如下：

$$\text{平均开销} = \frac{\text{本地包发送次数} + \text{转发包发送次数}}{\text{本地包个数}} \tag{6.1}$$

平均深度的计算就需要先获知汇聚树的拓扑结构，这可以通过监视父节点选择的变化获得。投递率的计算除了要监视转发事件之外，还需要监视根节点的包接收事件以得到投递成功的信息。根据上述方法对 10 个节点进行仿真，对比了使用标准 LE 和 4BITLE 的两种情况，得到表6.3所示结果：

表 6.3: 仿真结果

	标准 LE	4BITLE	增大消息缓存的 4BITLE
转发包发送次数	3294	3002	3326
本地包发送次数	1292	1272	1205
本地包总个数	121	120	124
投递包个数（有重复）	252	334	123
投递包个数（去重复）	109	115	117
开销	37.90	35.62	36.54
本均深度	2.0	2.9	3.5
投递率	90.08%	95.83%	94.35

从结果中可以看出，包发送的次数几乎是根节点实际接收到的包数的 20~40 倍，这是由于仿真使用的节点间连接质量（见表6.1）设定的较差，导致节点间不断发生重传所造成的，这也是开销比平均深度大 10~20 倍左右的原因。但即便是在通信质量如此差的网络中，CTP 协议还是可以成功地建立起多跳汇聚树，并保持了在 90% 以上的较高的投递率，说明 CTP 协议是比较可靠的。

对比标准 LE 和 4BITLE 的数据可以发现，4BITLE 即使在本均深度不如标准 LE 的情况下投递率还是明显高于标准 LE 的投递率，这是由于 4BITLE 使用了各层的信息得出的估计值比标准 LE 更精确，从而选择质量较差的连接的概率较小，丢包数也相应地减小。比较有重复的投递包个数和去重复的投递包个数可以发现，增大消息缓存后的 4BILTE 的包重复数（6 个）远小于原来的 4BITLE（219 个），这是由于消息缓存的增大使节点存储已收到的包数增加，在缓存中停留的时间增长，从而发现重复包的概率增大。

6.2 部署

6.2.1 自主开发节点简介

实际部署的节点使用的是自主开发的 npumote 平台节点，它使用微处理器 Atmega1281 和无线模块 RF230，配备了温湿度、光照等传感器，主要用于温室群的精准农业监控。

6.2.2 为 TinyOS 2.x 添加平台支持

由于 TinyOS 2.x 并没有直接支持我们的平台 npumote，因而要对它做一些修改。主要的修改有如下几个方面：

1. 修改 make 系统： 增加对 avrisp2 烧写器支持；增加 npumote 编译目标规则。
2. 修改硬件参数和连接方式： 修改串口的收发波特率；将 AM 层直接接到 UniqueLayerC 模块而不通过 IEEE154 网络层；增加 EEPROM 模块；修改 RF230 信道使程序在编译时可以通过增加 RF230_DEF_CHANNEL 标记设定信道值从 EEPROM 中读取；

将所有的改动以 GNU diff 格式记录到文件 `npumote.patch`。在 unix 环境中，使用者只须将 `npumote.patch` 放在 `tinyos-2.x` 同一个目录下，进入 `tinyos-2.x` 目录, 执行命令：

```
patch -p1 <../npumote.patch
```

即可使 TinyOS 2.x 支持 npumote 平台。

6.2.3 将程序烧写到节点中

在使用 CTP 协议的应用程序源代码目录 `apps/tests/TestNetwork` 下运行以下命令将编译生成二进制文件并烧写到节点中：

```
make npumote install.<节点号> avrisp2,<烧写器端口>
```

如果编译通过但烧写失败，则需要检查烧写器端口是否书写正确并且确认当前用户是否具有该端口的读写权限。

6.2.4 节点中程序的调试

由于节点的资源有限，程序的调试相对比较困难。传统的做法是使用节点上的 3 个 LED 灯提供节点的状态信息。比如在节点启动完毕事件触发时，在程序中指定将红色的 LED 灯开启；在接收或发送消息时，使 LED 灯闪烁等方法得知节点的工作信息。这种方法是最实时有效的，但也是原始和效率低下的。TinyOS 2.x 支持 printf 库的方法可通过串口向 PC 机发送调试信息，使调

试方便了不少，但仍存在诸多限制，如在串口开启之前的调试信息无法发送，不能设置断点和查看更改寄存器信息，修改程序后要重新烧写节点等问题。另外，节点也可以使用 JTAG 硬件调试器调试，由于我们自主开发节点对这种方法的工具链支持还不完全，故不在此详述。总之，节点中程序的调试还有许多工作可以做，可以作为以后研究工作的重点。

6.2.5 节点部署

节点号为 0 的节点在该应用程序中作为根节点使用，把它通过串口与上位机相连以便接收并处理汇聚上来的数据。其它节点的部署方法详见“无线传感器网络部署及其覆盖问题研究”[?]。

第七章　总结与展望

7.1 全文总结

汇聚协议是 TinyOS 中核心部分，理解汇聚协议对理解整个 TinyOS 架构有非常大的作用。本文使用自底向上的研究方法，循序渐进地分析了 TinyOS 2.x 中的 CTP 协议。主要做的工作如下：

1. 简述无线传感器网络体系结构，高度概括了节点的硬件特性、存在的限制和网络的组成形式。
2. 简练地介绍了 TinyOS 操作系统的功能、特点和工作原理与 nesC 语言。
3. 概述汇聚协议需要解决的问题，阐明 TinyOS 中 CTP 汇聚协议的总体架构，指出各部分之间的相互关联。
4. 按照自底向上的顺序逐层分析 CTP 协议的基本原理。从最底层的链路估计器部分入手，讨论了两个节点间的链路质量估计方法，详细分析了基于 LEEP 帧探测的标准 LE 链路估计器，并简要描述了另一种更高效的链路估计方法 4BITLE。
5. 对中间层路由选择部分进行分析。探讨了路由引擎如何在节点资源受限的情况下高效地选择父节点的重要性，阐明了它选择下一跳所使用的路径 ETX 路由选择策略，同时也分析了路由协议工作的时序。
6. 对上层的转发引擎进行分析。阐明了路由循环和包重复现象产生的原因，指出两者同时发生时对网络的不良影响，以及解决路由循环和抑制包重复的方法。另外解释了缓冲区分配的方法和缓冲区交换的设计意图。最后从本地包和转发包的发送这两方面指明转发引擎的工作流程。
7. 对使用 CTP 协议的 TinyOS 应用程序的仿真作了详细的介绍，讲述了如何让节点在 TOSSIM 仿真中通信的方法，提出了一种可视化仿真的可行方案。接着分析仿真结果，给出仿真节点最终形成的拓扑结构。然后提出三个衡量汇聚协议性能的指标：开销、平均深度和投递率，并根据这几个指标对比使用标准 LE 估计器和 4BITLE 的性能差异，并简单分析了造成差异的原理。
8. 描述将 TinyOS 移植到自主开发的节点平台 npumote 的方法，介绍了一些调试的心得，并对节点部署提出一些看法。

7.2 对未来工作的展望

根据本文的分析，可以发现 TinyOS 中的 CTP 协议已经可以很好的工作，但也存在不少可以改进的地方，可以作为未来研究的重点，概括起来主要有如下几个方面：

1. 可靠性问题。CTP 协议对可靠性没有完全的保障。数据收集节点在向汇聚树中发送一个数据包后，不能得知该包是否被根节点接收和处理。因此，如果有实际应用要求绝对的可靠性，则 CTP 协议将不适用。如何以较小的代价为 CTP 增加应答机制以实现通信的可靠性，这是一个值得研究的问题。
2. 数据分片问题。CTP 协议并不对发送的数据分片，如果有应用要求节点一次性收集发送的数据量较大（如视频采集），则 CTP 协议组成的网络吞吐率不高，很容易发生阻塞丢包现象。而对带宽有限的网络来说，超过带宽大小的数据包将无法发送。为 CTP 协议增加数据包分片机制是否能为网络性能带来提升，这也值得探讨。
3. 能耗问题。CTP 协议没有考虑节点的能量剩余，这样可能使汇聚树中的一些通信量较大的节点率先从网络中消亡，从而减短了整个网络的生存期。如果将能量消耗问题考虑进去，在保证能通信的前提下平衡各个节点能耗，可能会使网络的生存时间上一个台阶。
4. 大规模节点通信。在节点数量大，分布过密的情况下，使用 CTP 协议的节点在启动初期网络中的广播通信量会异常的大，信道使用冲突现象很严重，从而影响节点正确地建立路由。可以寻找一种方法将这部分通信量按时间错开，以提高信道的利用率。另外，CTP 协议由于资源限制使用的路由表较小，因此在邻居节点多的情况下可能连接质量最好的节点没有机会被选入路由表，从而使路由选择不能达到最优。如何使连接质量最好的节点必定能出现在路由表中，这也是具有研究的价值的。
5. 以数据为中心进行数据融合。CTP 协议中的转发节点并不对转发数据作任何更改。但是在某些应用中，地理位置邻近的节点收集到的数据往往具有相关性，如果能在转发节点进行数据融合或数据筛选，将可以减少网络中的通信量，从而延长网络的生存时间。

致　谢

首先要感谢我的导师李士宁教授。论文是在我的导师李老师的悉心指导下完成的。在论文的进展中，李老师提供了实验平台和学习资料，并且做了不少指点，对论文写作提出了很多宝贵意见。李老师严谨的治学态度以及丰富的实践经验将是我以后学习和工作的动力和楷模。在此谨向我的导师李士宁教授表示衷心的感谢。

同时要感谢传感器网络教研室李志刚老师和师兄师姐们，他们为我创造了良好的学习环境。他们提供的各种资料对我帮助很大，特别是林黛娣师姐在相同领域所作工作留下的资料使我受益匪浅。

另外要感谢 UC Berkley 的 Philip Levis，USC 的 Omprakash Gnawali 等人，是他们创造了 TinyOS 这个自由和优秀的 WSN 操作系统并提供了完整详细的文档；感谢为 GNU/Linux 贡献代码的黑客们，这个自由免费的平台为我完成毕设提供了不少便利；感谢清华大学王磊博士，他创作的 LaTeX 模板使我的论文的排版得以顺利完成。

最后感谢我的家人对我一如既往的关心和支持，感谢即将出世的侄儿给我带来精神上莫大的鼓舞。

毕业设计小结

这次毕业设计是对我四年本科学习的一个总结，涉及了操作系统、计算机网络、体系结构、组成原理等课程的知识，并且要求自己动手实践，这对我来说是一次全面的考验。一开始对于 TinyOS 和 nesC 语言完全是陌生的，对组件化设计的概念理解也不够深入。接着是各种安装和配置问题，只能通过官方的教程和 TinyOS 的邮件列表查询，信息的来源比较少。由于 nesC 并不是一种广泛使用的语言，因此各种相关的工具比较少，比如没有优秀的可视化工具可用，从而导致阅读 TinyOS 代码相对比较困难：为了找一个命令的实现，需要顺着配置文件层层挖掘，有时甚至要深入十几层才能找到具体实现的代码，后来通过自定义配置编辑器以及查找辅助工具才略微提高了一些效率，这一过程颇为坎坷。

本次毕业设计中印象比较深刻的是节点上程序的调试，节点没有足够的资源用于支持断点，甚至获知节点的当前运行状态也是相当困难的，通过串口的调试信息并不一定是实时的，因而只能通过节点上的 3 个 LED 灯得知准确状态信息，这是以后可以改进的地方。

经历了本次毕设，我对无线传感器网络有了一定的了解，积累了一些实际经验，对以后研究生阶段的学习目标也更加明确了。