WSN实验报告

孙子平 2015013249 车行 2015013241 李在弦 2015080121

1 题目一：多跳WSN数据采集

1.1 关于

1.1.1 项目特色

- 自由组网：可自由组网成树状的拓扑结构，支持任意数目节点组网，可视化支持任意数目节点
- 可靠传输：几乎0丢包率，即使节点掉线一阵子，重连之后，依旧不会丢包
- 实时交互的可视化界面：基于Web的可视化，实时统计丢包数目、重包数目等，统计图表可交互显示

1.1.2 项目结构

项目的根目录位于`task1`下，包括以下文件、文件夹：

- `Node/`：Telosb节点的代码，我们所有的节点都采用同一份代码
  - `ForwarderC.nc`：泛型传输模块，负责在父子节点之间建立可靠传输
  - `SensorC.nc`：传感器模块，负责收集数据
  - `AppC.nc`：顶层模块
- `Visualizer/`：数据可视化的Web服务端
  - `main.py`：服务端代码
- `nodes.csv`：配置文件，用以为不同的节点产生不同的编译选项

1.1.3 使用方式

首先编辑`nodes.csv`，格式如下方的表格。如果节点是`Sensor`，则会收集传感器数据；如果节点是`Base Station`，则其上游通信不是采用无线而是采用串口。

ID	Father ID	Sensor	Base Station
0	0	0	1
1	0	1	0

ID	Father ID	Sensor	Base Station
2	1	1	0

之后可以烧录程序，运行可视化界面：

```
cd task1
# 编译节点代码，<node_id>为节点的ID
make compile,<node_id>
# 烧录节点程序，<node_id>为节点的ID
make install,<node_id>
# 不停烧录节点程序直至成功，<node_id>为节点的ID
./FuckMake install,<node_id>
# 安装可视化的依赖
pip install -r Visualizer/requirements.txt
# 启用可视化，后面可跟串口参数，默认是serial@/dev/ttyUSB0:115200
python Visualizer/main.py
```

这时，可以浏览器打开`http://localhost:8050/`看到可视化的效果。这里3个节点的Sensor都开启了，所有的湿度传感器都有问题，而1号节点的光照传感器也有问题。

“刷新间隔”是指轮询服务器的间隔（实时改变），“显示时长”是下方各个表显示的x轴的范围（实时改变），“采样间隔”是每个节点的采样计时器触发的时间间隔（需要点击发送按钮改变）。

“手动刷新”用于触发一次轮询。“同步时间”会将下一匹收到的包作为新的基准时间，推测新的包的真实发送时间，应当在网络条件较好的情况下再点击。“清除数据”会清空显示的数据。

所有的图都可以缩放，查看细节数据。

`result.txt`是收到的最原始数据，注意我们始终用append模式打开这个文件。重复的包也会被记录下来。

多跳WSN数据采集可视化

如果收到的包的序列号很接近上一次的序列号时，我们会依照序列号计算**包丢失数**或**包重复数**；如果序列号差别很大，我们会认为节点重启了或离开网络很久，计入**重置数**，并重新同步时间，重新计算**真实采样间隔**。

真实采样间隔是通过系数为 0.90 的指数移动平均（EMA）计算的。

对于显示的数据，我们依照TinyOS的教程进行了一些处理。

刷新间隔 1s

1

显示时长 2min

2

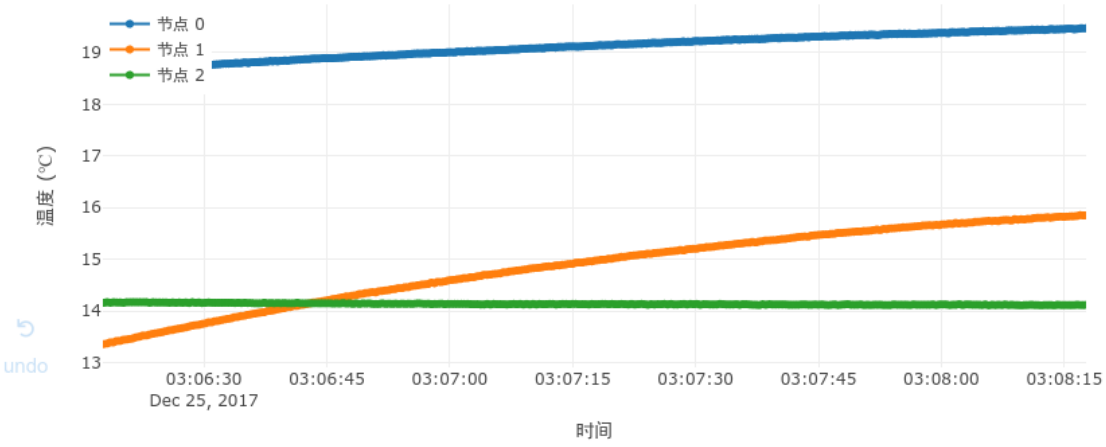
采样间隔 100ms

100 发送

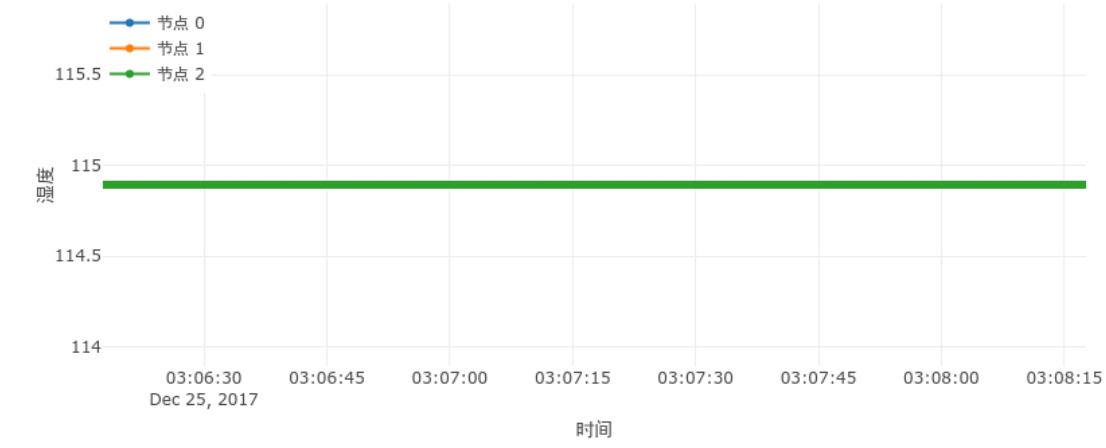
手动刷新 同步时间 清除数据

节点编号	0	1	2
包收到数	437	328	328
包丢失数	0	0	0
包重复数	0	0	0
重置数	0	0	0
真实采样间隔 (ms)	300	400	400

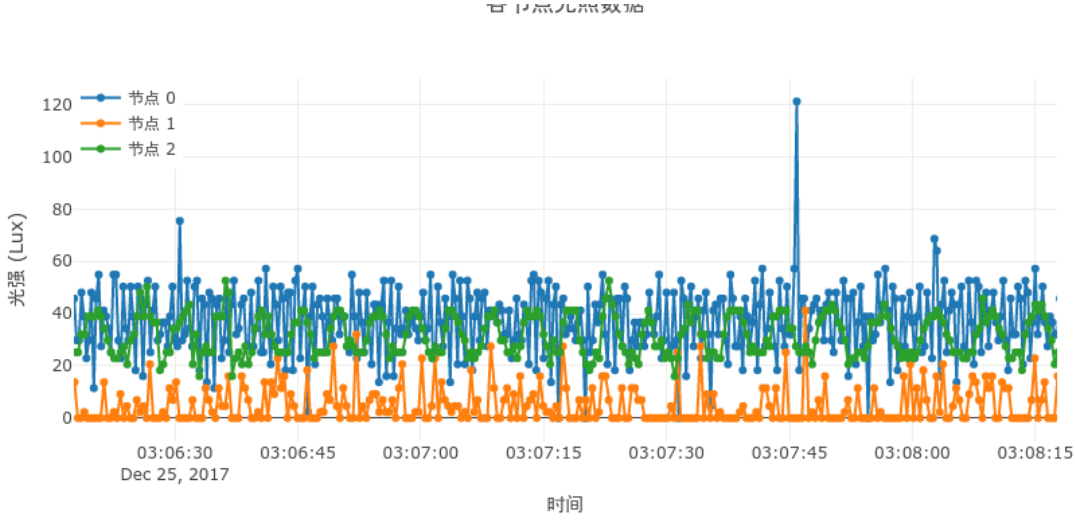
各节点温度数据



各节点湿度数据



各节点光照数据



1.2 实现

1.2.1 节点实现

可靠传输： 首先，我们发送的数据包分为两类，往根节点传的数据消息和往叶节点传的控制消息。前者默认拥有512的循环队列，而后者默认拥有32的缓冲区。而后我们借助Tinyos自带的PackageAcknowledgements借口确保1个数据包被成功发送，如果没成功发送则重试。整个发送等价滑窗为1的GBN。

组网： 组网我们采用静态的路由表，而后以编译参数的方式编译到程序中。数据发送我们都是指定接受者，而非广播。对于多个子节点的情况，我们是挨个发个去并确认收到的。

数据采集： 数据采集我们有两种方案，一种是等待全部采集完毕再采集下一匹（默认），另一种是有个采集队列，不必等待采集完成就发起下一次采集（解除`task1/Node/Makefile`中`RECORD_QUEUE`选项的注释即可启用）。实际证明，后者在采集速度上几乎等价于前者，呈现出来的时间则会有偏差，故不采用。实际节点采样频率在280~320ms左右。

1.2.2 可视化实现

我们采用的数据可视化框架是Dash by plotly。数据采集则用的是TinyOS自带的Python SDK。具体实现上，我们启动了额外的线程专门用于串口通信。

包数目的统计： 对于每个收到的包，我们会查看是否已经收到过这个节点的包。如果还未收到，我们则初始化这个节点的相关数据，同步时间，并开始计算“真实采样间隔”。如果已经收到，我们会检查包的序号。如果序号超前不多（16以内），我们会计算“包丢失数”；如果序号落后的不多（16以内），则会计入“包重复数”；如果序号变化实在是太大，则我们会计入“重置数”，并重新同步时间，重新计算“真实采样间隔”。

时间同步： 由于收到的包的时间戳，是相对于发送节点的开启时刻。而我们没有复杂的同步协议。所以时间同步就以收到的该节点的第一个包的时间为准。点击“同步时间”和“清除数据”，都会强制以下一个包收到的时间作为基准。

真实采样频率： 采样频率的估计是用指数移动平均，具体公式如下，其中，$\hat{t}_k$是收到第k个包时对真实采样间隔的估计，t_k是收到第k个包时计算得到的此刻采样间隔，r是系数，这里取0.9。

$$\hat{t}_k = r \cdot t_k + (1 - r) \cdot \hat{t}_{k-1}$$

2 题目二：多点协作Data Aggregation实验

2.1 关于

2.1.1 项目特色

- 自由组合：支持任意数目的节点，以任意方式分配任务
- 多种策略：可以选择是边收包边询问peers未收到的包还是等收完后再询问，可以选择中位数的算法

2.1.2 项目结构

项目的根目录位于`task2`下，包括以下文件、文件夹：

- `Master/`：模拟的助教节点，负责发送数据，接受答案并返回ACK
- `Slave/`：本项目真正的代码所在，负责协作计算
  - `CalculateC.nc`：负责计算的模块
  - `TransportC.nc`：负责接受数据、协商补齐丢失数据、汇总计算结果的模块
  - `SlaveAppC.nc`：顶层模块
  - `Makefile`：包含了若干的编译选项，用以改变策略
- `nodes.csv`：配置文件，用以为不同的节点产生不同的编译选项

2.1.3 使用方式

首先编辑`nodes.csv`，格式如下方的表格，Task是`:`分隔的一系列任务。

ID	Role	Task
0	Master	receive
28	Slave	median:collect

ID	Role	Task
29	Slave	max:min:backup
30	Slave	average:sum:backup
1000	Master	send

如果节点是Master，则任务可以为以下两种的组合：

- send：发送2000个数据
- receive：接受答案，返回ACK并向串口发送接收到的答案

如果节点是Slave，则任务可以为以下七种的组合：

- max, min, sum, average, median：完成对应的计算任务
- collect：收集所有的答案并发送给Master
- backup：备份收到的数据，用于提供补齐数据

其中列表中必须有且只有一个任务是receive的Master，有且只有一个任务是send的Master，有且只有一个任务是collect的Slave。由于内存限制，Slave不能同时有median和backup任务。如果一个Slave没有计算任务，则它不会去申请补齐数据。如果多个Slave有同样的计算任务，那么最先送到collect节点的结果会被采用。

然后，`Slave/Makefile`包含了两个编译选项：

- `QUERY_AFTER_SENDING`：开启时，会在全部数据接受完毕之后，再申请补齐数据；否则一边接受数据一边协商补齐数据
- `MEDIAN_AFTER_SENDING`：开启时，会在全部数据接受完毕之后，采用一种类似快排的$O(n)$算法，找到中位数；否则则一边接受数据一边采用有DMA加速的类似插入排序的方式找中位数

最后可以烧录程序：

```
cd task2
# 编译节点代码，<node_id>为节点的ID
make compile,<node_id>
# 烧录节点程序，<node_id>为节点的ID
make install,<node_id>
# 不停烧录节点程序直至成功，<node_id>为节点的ID
./FuckMake install,<node_id>
```

经过实验，在全部数据接收完毕之后再进行协商补齐数据和寻找中位数性能更好，因为这降低了丢包率。此外类似快排的中位数算法实际上能在几ms乃至更低的级别上完成计算。

而将所有计算任务放在collect节点也能获得最高的性能，这样减少了丢包申请补齐的数目。

2.2 实现

以下会出现多个常量，这些常量位于`Slave/message.h`中。

确认丢包： 当节点收到数据，且数据的序号小于等于5（`MINIMUM_START_SEQ`）的时候，开始接受数据。如果收到的下一个包序号是比所有以前的包序号大的，则认为这之间的缺失的序号都是丢失的包，存入队列。如果超过50ms（`NEXT_SEQ_TIMEOUT`），则会强制把下一个序号加入缺失的序号中。

协商补齐丢包： 当节点有丢包的时候，会每隔100ms（`TEST_SEND_INTERVAL`）向所有的backup节点申请补齐这个包。如果backup节点有该包则会返回，否则不做应答。如果节点收到了要补齐的包，则会立刻尝试补齐下一个包。

中位数计算： 我们考虑了插入排序，快速排序和堆排序。插入排序拥有$O(n)$的插入复杂度，和$O(1)$的查询复杂度；快速排序拥有$O(1)$的插入复杂度（不做操作），和$O(n)$的查询复杂度；堆排序拥有$O(\log n)$的插入复杂度和$O(n \log n)$的查询复杂度。综合之下我们排除了堆排序，采用了插入排序和快速排序。

DMA加速： 实现插入排序的时候，复制数据占耗掉了大量的时间导致丢包率很高。故而我们考虑了采用DMA（Direct Memory Access，无需CPU介入的内存读写）加速整个排序算法。经查询，MSP430再启用DMA复制数据的时候，CPU只有原先20%的工作效率。最后，我们会在复制数据的数目小于16（`DMA_MEMMOVE_THRESHOLD`）的时候采用`memmove`，否则采用DMA加速。实验表明，采用DMA，丢包率有了显著的下降，证明CPU被更高效地利用了。

汇总结果： 我们使用PackageAcknowledgements确认计算结果发送至collect节点。collect节点等到所有的结果汇总完毕时，每隔100ms（`TEST_SEND_INTERVAL`）向Master的receive节点发送答案，直到收到该节点的ACK包才会停止发送。

3 项目分工

- 孙子平：题目1的`ForwarderC`（可靠传输模块）和题目2的`TransportC`（协商补齐丢包、汇总结果）
- 车行：题目1的`SensorC`（传感器模块）和题目2的`CalculateC`（计算模块）
- 李在弦：题目1的`Visualizer/main.py`（数据可视化）