Traffic Control

TEP:	137
Group:	Core Working Group
Type:	Documentary
Status:	Draft
TinyOS-Version:	2.x
Author:	David Moss, Mark Hays, and Mark Siner
Draft-Created:	3-Sept-2009
Draft-Version:	1.1
Draft-Modified:	2009-10-01
Draft-Discuss:	TinyOS Developer List <tinyos-devel at mail.millennium.berkeley.edu>

Abstract

This memo proposes traffic control interfaces to be provided by an optional traffic control layer integrated at the highest levels of communication stacks. These traffic control mechanisms are targeted to help improve acknowledgment success rate, energy efficiency, fairness, and routing reliability on any wireless platform. The available reference implementation is a platform independent radio stack layer designed to consume a very small memory footprint.

1. Introduction

As the traffic rate of a wireless sensor network increases, the probability of collision, dropped packets, and missed acknowledgments also increases, even with sophisticated CSMA/CA implementations.

It is important, especially in the case mesh networks, for packets to be delivered reliably and acknowledgments to be returned successfully on a hop-by- hop basis. One method to improve reliability is to reduce the rate of transmissions from each node within the network.

Traffic Control has been in use for years in many different wired and wireless applications[1]_,[2]_,[3]_. TinyOS already has traffic control mechanisms integrated directly into some networking libraries, such as CTP[4]_ and Dissemination[5]_. The use of Trickle[6]_ algorithms, also used within CTP and Dissemination, further reduces the rate of traffic throughout a network to improve delivery performance and prevent livelock. There has yet to be a centralized method of traffic control that throttles traffic generated from any component of a user's application.

The traffic control interfaces proposed in this TEP are very basic, and are intended to support many different traffic control implementations. Two interfaces assist the application layer in controlling behavior: TrafficControl and TrafficPriority.

The reference implementation presented here is integrated as a optional and generic radio stack layer (providing a Send and using a SubSend interface) and uses acknowledgments to dynamically adjust the transmit throttle. Other traffic control implementations could employ more sophisticated techniques to control throughput, but likely at the cost of a larger memory footprint.

The ultimate goal is to allow developers to use mesh networking protocols and/or their own protocols without having to worry about implementing any kind of traffic control timer mechanism for each separate component.

2. Desired Behavior

Ideally, a traffic control layer SHOULD attempt to balance the rate of transmissions from a single node with the channel throughput capacity. This implies an adaptive control mechanism. If the channel is busy, nodes should add delay between packets to let other nodes transmit. Similarly, if the channel is not busy, a node should be allowed access to the channel more often to prevent inefficient channel downtime. Traffic control SHOULD NOT listen to the channel for long periods of time to determine the appropriate access rates, because that defeats the purpose of low power communications layers used elsewhere.

The traffic control implementation SHOULD have the option to be activated or deactivated on a system-wide level as well as a packet level. This allows for individual high or low priority packets. Traffic control SHOULD be deactivated by default, until the application or networking layers explicitly enable it.

Finally, the traffic control mechanism SHOULD be small in code size to fit on the limited program memory available on most wireless platforms. There SHOULD NOT be additions or modifications to a packet's metadata structure that enables or disables traffic control on a per-packet basis; instead, per-packet priorities SHOULD be performed with a request/call back procedure. This keeps RAM requirements low and can be optimized out at compile time if those functions are not used.

We also recommend any traffic control layer be implemented as an optional compile time add-on to a core radio stack or within the ActiveMessageC platform communication stack definition. This allows applications that do not require traffic control to remove its memory footprint from the system.

3. TrafficControlC Component Signature

The signature of TrafficControlC is RECOMMENDED as follows:

```
configuration TrafficControlC {
  provides {
    interface Send;
    interface TrafficControl;
    interface TrafficPriority[am_id_t amId];
  }

  uses {
    interface Send as SubSend;
  }
}
```

The Send interface provided on top and SubSend interface used underneath allow the TrafficControlC component to be integrated as a generic layer within any radio stack.

4. TrafficControl Interface

The TrafficControl interface allows the application layer to enable or disable traffic control from a system-wide level. It also allows an application to set and get the current delay between packets. For most systems, we expect that the setDelay() and getDelay() commands may not be used often and will most likely get optimized out at compile time; however, some systems may care to explicitly increase or decrease the delay between packets or collect statistics on how the traffic control layer is performing.

The TEP proposes the following TrafficControl interface:

```
interface TrafficControl {

  command void enable(bool active);

  command void setDelay(uint16_t delay);

  command uint16_t getDelay();

}
```

5. TrafficPriority Interface

The TrafficPriority interface is parameterized by active message ID. It is a simple request / call back interface that allows components in the application layer to configure individual packets for priorities on a scale from 0 (lowest priority, default) to 5 (highest priority, get the packet out immediately). There are several advantages to this call back method. Metadata does not need to be added to the end of every message_t. Additionally, a component that captures a requestPriority(...) event is not required to adjust the priority as it would if the event returned a value.

When a packet enters the traffic control layer, and traffic control is enabled, the TrafficPriority interface MUST signal out the event requestPriority(...). This event, with all the extra information it provides, allows the application layer to decide whether the packet is a high priority packet or not. Calling the setPriority(uint8_t priority) command within the requestPriority(...) event MAY adjust the traffic control mechanisms applied to the current packet. To aid in the definition of priority, two definitions are available in TrafficControl.h:

```
enum {
  TRAFFICPRIORITY_LOWEST = 0,
  TRAFFICPRIORITY_HIGHEST = 5,
};
```

It is up to the traffic control implementation to define the meaning of each priority level. In the reference implementation, a priority of 0 is the default low priority level that employs the full traffic control delays. Anything above 0 in the reference implementation is considered to be at the highest priority.

If no areas of the application layer care to change the packet's priority, a default event handler will capture the requestPriority(...) event and do nothing. This would result in all packets being sent at a low priority with full traffic control mechanisms enforced.

The TEP proposes the following TrafficPriority interface, to be provided as an interface parameterized by AM type:

```
  interface TrafficPriority {

    event void requestPriority(am_addr_t destination, message_t \*msg);

    command void setPriority(uint8_t priority);

  }
```

6. Reference Implementation

An implementation of the proposed traffic control layer can be found in the CCxx00 radio stack in tinyos-2.x-contrib/blaze/tos/chips/ccxx00_addons/trafficcontrol, with interfaces located in tinyos-2.x-contrib/blaze/tos/chips/ccxx00_single/interfaces and a dummy implementation located in tinyos-2.x-contrib/blaze/tos/chips/ccxx00_single/traffic.

In this implementation, the default core radio stack (ccxx00_single) includes an empty stub for traffic control. Users that wish to include the traffic control implementation in their systems simply override the default stub component with the ccxx00_addons/trafficcontrol directory.

The reference implementation works as follows. All nodes start with a default of 4 seconds between each packet. Changes are made to the time between outbound packets only when a unicast packet is sent with the request for acknowledgment flag set. The reception of an acknowledgment is used as a basic indicator of channel activity. For each acknowledgment received, the amount of time between packets is decreased so the next packet will get sent faster. For each dropped acknowledgment, the amount of time between packets increases, causing the next packet to be sent later.

When the transmission rate reaches a boundary (1 second per packet per node fastest, 10 seconds per packet per node slowest), it is reset to the default rate of 4 seconds per packet per node. This prevents nodes from unfairly capturing the channel.

Testing this traffic control layer in a congested test bed setting of 16 nodes with multiple hidden terminals resulted in the acknowledgment success rate moving from 27-50% without traffic control to 90-100% with traffic control. The memory footprint increased by 260 bytes ROM / 16 bytes RAM with the inclusion of the traffic control layer.

5. Author Addresses

David Moss
Rincon Research Corporation
101 N. Wilmot Suite 101
Tucson AZ 85750
email: mossmoss at gmail dot com

Mark Hays
Rincon Research Corporation
101 N. Wilmot Suite 101
Tucson AZ 85750
email: mhh at rincon dot com

Mark Siner
Rincon Research Corporation
101 N. Wilmot, Suite 101
Tucson, AZ 85750
email: mks at rincon dot com

6. Citations

[1] Bret Hull, Kyle Jamieson, Hari Balakrishnan. "Mitigating Congestion in Wireless Sensor Networks." In the Proceedings of the ACM Sensys Conference 2004

[2] Wan, C.-Y., Eisenman, S., and Campbell, A. "CODA: Congestion Detection and Avoidance in Sensor Networks." In the Proceedings of the ACM Sensys Conference 2003

[3] Woo, A., and Culler, D. "A Transmission Control Scheme for Media Access in Sensor Networks." In ACM MOBICOM 2001

[4] Rodrigo Fonseca, Omprakash Gnawali, Kyle Jamieson, Sukun Kim, Philip Levis, and Alec Woo.. "TEP123: Collection Tree Protocol"

[5] Philip Levis and Gilman Tolle. "TEP118: Dissemination of Small Values."

[6] Philip Levis, Neil Patel, David Culler, and Scott Shenker. "Trickle: A Self-Regulating Algorithm for Code Maintenance and Propagation in Wireless Sensor Networks." In Proceedings of the First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI 2004).