Collection

TEP:	119
Group:	Net2 Working Group
Type:	Documentary
Status:	Final
TinyOS-Version:	> 2.1
Author:	Rodrigo Fonseca, Omprakash Gnawali, Kyle Jamieson, and Philip Levis
Draft-Created:	09-Feb-2006
Draft-Discuss:	TinyOS Developer List <tinyos-devel at mail.millennium.berkeley.edu>

Note

This memo documents a part of TinyOS for the TinyOS Community, and requests discussion and suggestions for improvements. Distribution of this memo is unlimited. This memo is in full compliance with TEP 1.

Abstract

The memo documents the interfaces, components, and semantics used by the collection protocols in TinyOS 2.x. Collection provides best-effort, multihop delivery of packets to one of a set of collection points. There may be multiple collection points in a network, and in this case the semantics are *anycast* delivery to at least one of the collection points. A node sending a packet does not specify which of the collection points the packet is destined to. The union of the paths from each node to one or more of the collection points forms a set of trees, and in this document we assume that collection points are the roots of these trees.

1. Introduction

Collecting data at a base station is a common requirement of sensor network applications. The general approach used is to build one or more collection trees, each of which is rooted at a base station. When a node has data which needs to be collected, it sends the data up the tree, and it forwards collection data that other nodes send to it. Sometimes, depending on the form of data collection, systems need to be able to inspect packets as they go by, either to gather statistics, compute aggregates, or suppress redundant transmissions.

Collection provides best-effort, multihop delivery of packets to one of a network's tree roots: it is an *anycast* protocol. The semantics are that the protocol will make a reasonable effort to deliver the message to at least one of the roots in the network. By

picking a parent node, a node implementing the collection protocol inductively joins the tree its parent has joined. Delivery is best effort, and there can be duplicates delivered to one or more roots. Collection provides no ordering or real-time guarantees, although specific implementations may extend the basic functionality to do so.

Given the limited state that nodes can store and a general need for distributed tree building algorithms, collection protocols encounter several challenges. These challenges are not unique to collection protocols. Instead, they represent a subset of common networking algorithmic edge cases that generally occur in wireless routing:

- Loop detection, for when a node selects one of its descendants as a next hop.
- Duplicate suppression, detecting and dealing with lost acknowledgments that can cause packets to replicate in the network, wasting capacity.
- Link estimation, evaluating the link quality to single-hop neighbors.
- Self-interference, preventing forwarding packets along the route from introducing interference for subsequent packets.

While collection protocols can take a wide range of approaches to address these challenges, the programming interface they provide is typically independent of these details. The rest of this document describes a set of components and interfaces for collection services.

2. Collection interfaces

A node can perform four different roles in collection: sender (or source), snooper, in-network processor, and receiver (or root). Depending on their role, the nodes use different interfaces to interact with the collection component.

The collection infrastructure can be multiplexed among independent applications, by means of a collection identifier. The collection identifier is used to identify different data traffic at the sender, intermediate-nodes, or the receiver, much like port number in TCP. All data traffic, regardless of the collection identifier, use the same routing topology.

The nodes that generate data to be sent to the root are *senders*. Senders use the Send interface [1] to send data to the root of the collection tree. The collection identifier is specified as a parameter to Send during instantiation.

The nodes that overhear messages in transit are *snoopers*. The snoopers use the Receive interface [1] to receive a snooped message. The collection identifier is specified as a parameter to Receive during instantiation.

The nodes can process a packet that is in transit. These in-network *processors* use the Intercept interface to receive and update a packet. The collection identifier is specified as a parameter to Intercept during instantiation. The Intercept interface has this signature:

```
interface Intercept {
  event bool forward(message_t* msg, void* payload, uint8_t len);
}
```

Intercept has a single event, Intercept.forward(). A collection service SHOULD signal this event when it receives a packet to forward. If the return value of the event

is FALSE, then the collection layer MUST NOT forward the packet. The Intercept interface allows a higher layer to inspect the internals of a packet and suppress it if needed. Intercept can be used for duplicate suppression, aggregation, and other higher-level services. As the handler of Intercept.forward() does not receive ownership of the packet, it MUST NOT modify the packet and MUST copy data out of the packet which it wishes to use after the event returns.

Root nodes that receive data from the network are *receivers*. Roots use the Receive interface [1] to receive a message delivered by collection. The collection identifier is specified as a parameter to Receive during instantiation.

The set of all roots and the paths that lead to them form the collection routing infrastructure in the network. For any connected set of nodes implementing the collection protocol there is only one collection infrastructure, *i.e.*, all roots in this set active at the same time are part of the same infrastructure.

The RootControl interface configures whether a node is a root:

```
interface RootControl {
  command error_t setRoot();
  command error_t unsetRoot();
  command bool isRoot();
}
```

The first two commands MUST return SUCCESS if the node is now in the specified state, and FAIL otherwise. For example, if a node is already a root and an application calls RootControl.setRoot(), the call will return SUCCESS. If setRoot() returns SUCCESS, then a subsequent call to isRoot() MUST return TRUE. If unsetRoot() returns SUCCESS, then a subsequent call to isRoot() MUST return FALSE.

3 Collection Services

A collection service MUST provide one component, CollectionC, which has the following signature:

```
configuration CollectionC {
  provides {
    interface StdControl;
    interface Send[uint8_t client];
    interface Receive[collection_id_t id];
    interface Receive as Snoop[collection_id_t];
    interface Intercept[collection_id_t id];
    interface RootControl;
    interface Packet;
    interface CollectionPacket;
  }
  uses {
    interface CollectionId[uint8_t client];
  }
}
```

CollectionC MAY have additional interfaces. All outgoing invocations (commands for uses, events for provides) of those interfaces MUST have default functions. Those

default functions enable CollectionC to operate properly even when the additional interfaces are not wired.

Components SHOULD NOT wire to CollectionC.Send. The generic component CollectionSenderC (described in section 3.1) provides a virtualized sending interface.

Receive, Snoop, and Intercept are all parameterized by collection_id_t. Each collection_id_t corresponds to a different protocol operating on top of collection, in the same way that different am_id_t values represent different protocols operating on top of active messages. All packets sent with a particular collection_id_t generally SHOULD have the same payload format, so that snoopers, intercepters, and receivers can parse them properly.

ColletionC MUST NOT signal Receive.receive on non-root nodes. CollectionC MUST signal Receive.receive on a root node when a unique (non-duplicate) data packet successfully arrives at that node. It MAY signal Receive.receive when a duplicate data packet successfully arrives. If a root node calls Send, CollectionC MUST treat it as it if were a received packet. Note that the buffer swapping semantics of Receive.receive, when combined with the pass semantics of Send, require that CollectionC make a copy of the buffer if it signals Receive.receive.

If CollectionC receives a data packet to forward and it is not a root node, it MAY signal Intercept.forward. CollectionC MAY signal Snoop.receive when it hears a packet which a different node is supposed to forward. For any given packet it receives, CollectionC MUST NOT signal more than one of the Snoop.receive, Receive.receive, and Intercept.forward events.

RootControl allows a node to be made a collection tree root. CollectionC SHOULD NOT configure a node as a root by default.

Packet and CollectionPacket allow components to access collection data packet fields [1].

3.1 CollectionSenderC

Collection has a virtualized sending abstraction, the generic component CollectionSenderC:

```
generic configuration CollectionSenderC(collection_id_t collectid) {
  provides {
    interface Send;
    interface Packet;
  }
}
```

This abstraction follows a similar virtualization approach to AMSenderC [1], except that it is parameterized by a collection_id_t rather than an am_id_t. As with am_id_t, every collection_id_t SHOULD have a single packet format, so that receivers can parse a packet based on its collection ID and contents.

4. Implementation

Implementations of collection can be found in `tinyos-2.x/tos/lib/net/ctp` and `tinyos-2.x/tos/lib/net/lqi`. The former is the Collection Tree Protocol

(CTP), described in TEP 123 [2]. The latter is a TinyOS 2.x port of MultihopLqi, a CC2420-specific collection protocol in TinyOS 1.x.

5. Author Addresses

Rodrigo Fonseca
473 Soda Hall
Berkeley, CA 94720-1776

phone - +1 510 642-8919
email - rfonseca@cs.berkeley.edu

Omprakash Gnawali
Ronald Tutor Hall (RTH) 418
3710 S. McClintock Avenue
Los Angeles, CA 90089

phone - +1 213 821-5627
email - gnawali@usc.edu

Kyle Jamieson
The Stata Center
32 Vassar St.
Cambridge, MA 02139

email - jamieson@csail.mit.edu

Philip Levis
358 Gates Hall
Computer Science Laboratory
Stanford University
Stanford, CA 94305

phone - +1 650 725 9046
email - pal@cs.stanford.edu

6. Citations

[1] TEP 116: Packet Protocols.
[2] TEP 123: The Collection Tree Protocol (CTP).