Collection

TEP:	119
Group:	Net2 Working Group
Type:	Documentary
Status:	Draft
TinyOS-Version:	2.x
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 collection protocol in TinyOS 2.x. Collection provides a best-effort, multihop delivery of packets to the root of *a* tree. There may be multiple roots in a network, and in this case the semantics implemented are of *anycast* delivery to at least one of the roots. A node sending a packet does not specify which root the packet is destined to.

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.

When a network has multiple base stations that act as *root* nodes, rather than one tree, it has a *forest* of trees. By picking a parent node, a collection protocol implicitly joins one of these trees. Collection provides a best-effort, multihop delivery of packets to one of a network's tree roots: it is an *anycast* protocol. The semantics is that the protocol will make a reasonable effort to deliver the message to at least one of the roots in the network. There are however no guarantees of delivery, and there can be duplicates delivered to one or more roots. There is also no ordering guarantees.

Given the limited state that nodes can store and a general need for distributed tree building algorithms, simple 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 occur in this protocol family:

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

The rest of this document describes a set of components and interfaces for a collection service outlined above.

2. Collection interfaces

A node can perform four different roles in collection: producer, consumer, snooper, and in-network processor. Depending on their role, the nodes use different interfaces to interact with the collection component.

A consumer is a root of a tree. 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.

A node is configured to become a root by using the RootControl interface. RootControl.setRoot() MUST make the current node a root of the the collection infrastructure. RootControl.unsetRoot() MUST make the current root no longer a root in the collection infrastructure. Both calls are idempotent. RootControl.setRoot() MAY be called on a node that is already a root, to no effect. RootControl.unsetRoot() MAY be called on a node that is not a root:

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

The collection infrastructure can be multiplexed among independent applications, by means of a *collection identifier*. It is important to note that the *data* traffic in the protocol is multiplexed, while the *control* traffic is not.

The nodes that generate data to be sent to the root are *producers*. The producers 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.

Root nodes that receive data from the network are *consumers*. The consumers 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 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 are in transit. These in-network *processors* use the Intercept interface [1] to receive and update a packet. The collection identifier is specified as a parameter to Intercept during instantiation.

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, but they MUST have default functions on all outgoing invocations (commands for uses, events for provides) of those interfaces so that it can operate properly if they 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 have the same payload format, so that snoopers, intercepters, and receivers can parse it properly.

Receive.receive MUST NOT be signaled on non-root nodes. CollectionC MAY signal Receive.receive on a root node when a data packet successfully arrives at that node. 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.

If CollectionC receives a data packet that a different node is supposed to forward, it MAY signal Snoop.receive.

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

An implementation of this TEP can be found in `tinyos-2.x/tos/lib/net/ctp` and `tinyos-2.x/tos/lib/net/le`, in the CTP protocol. It is beyond the scope of this document to fully describe CTP, but we outline its main components. CTP will be described in an upcoming TEP [2]. This implementation is a reference implementation, and is not the only possibility. It consists of three major components, which are wired together to form a CollectionC: LinkEstimatorP, CtpTreeRoutingEngineP, and CtpForwardingEngineP.

This decomposition tries to encourage evolution of components and ease of use through modularization. Neighbor management and link estimation are decoupled from the routing protocol. Furthermore, the routing protocol and route selection are decoupled from the forwarding policies, such as queueing and timing.

4.1 LinkEstimatorP

LinkEstimatorP estimates the quality of link to or from each neighbor. Link estimation can be done in a variety of ways, and we do not impose one here. It is decoupled from the establishment of routes. There is a narrow interface -- LinkEstimator -- between the link estimator and the routing engine. The one requirement is that the quality returned is standardized. A smaller return value from LinkEstimator.getQuality(), LinkEstimator.getforwardQuality(), LinkEstimator.getReverseQuality() MUST imply that the link to the neighbor is estimated to be of a higher quality than the one that results in a larger return value. The range of value SHOULD be [0,255] and the variation in link quality in that range SHOULD be linear. Radio provided values such as LQI or RSI, beacon based link estimation to compute ETX, or their combination are some possible approaches to estimating link qualities.

LinkEstimatorP MAY have its own control messages to compute bi-directional link qualities. LinkEstimatorP provides calls (txAck(), txNoAck(), and clearDLQ()) to update the link estimates based on successful or unsuccessful data transmission to the neighbors.

The user of LinkEstimatorP can call insertNeighbor() to manually insert a node in the neighbor table, pinNeighbor() to prevent a neighbor from being evicted, and unpinNeighbor() to restore eviction policy:

```
typedef uint16_t neighbor_table_entry_t

LinkEstimatorP {
  provides {
    interface StdControl;
    interface AMSend as Send;
    interface Receive;
    interface LinkEstimator;
    interface Init;
    interface Packet;
    interface LinkSrcPacket;
  }
}

interface LinkEstimator {
  command uint8_t getLinkQuality(uint16_t neighbor);
  command uint8_t getReverseQuality(uint16_t neighbor);
  command uint8_t getForwardQuality(uint16_t neighbor);
  command error_t insertNeighbor(am_addr_t neighbor);
  command error_t pinNeighbor(am_addr_t neighbor);
  command error_t unpinNeighbor(am_addr_t neighbor);
  command error_t txAck(am_addr_t neighbor);
  command error_t txNoAck(am_addr_t neighbor);
```

```
  command error_t clearDLQ(am_addr_t neighbor);
  event void evicted(am_addr_t neighbor);
}
```

4.2 CtpRoutingEngineP

CtpRoutingEngineP is responsible for computing routes to the roots of a tree. In traditional networking terminology, this is part of the control plane of the network, and is does not directly forward any data packets, which is the responsibility of CtpForwardingEngine. The main interface between the two is UnicastNameFreeRouting.

CtpRoutingEngineP uses the LinkEstimator interface to learn about the nodes in the neighbor table maintained by LinkEstimatorP and the quality of links to and from the neighbors. The routing protocol on which collection is implemented MUST be a tree-based routing protocol with a single or multiple roots. CtpRoutingEngineP allows a node to be configured as a root or a non-root node dynamically. CtpRoutingEngineP maintains multiple candidate next hops:

```
generic module CtpRoutingEngineP(uint8_t routingTableSize,
                                 uint16_t minInterval,
                                 uint16_t maxInterval) {
    provides {
        interface UnicastNameFreeRouting as Routing;
        interface RootControl;
        interface CtpInfo;
        interface StdControl;
        interface CtpRoutingPacket;
        interface Init;
    }
    uses {
        interface AMSend as BeaconSend;
        interface Receive as BeaconReceive;
        interface LinkEstimator;
        interface AMPacket;
        interface LinkSrcPacket;
        interface SplitControl as RadioControl;
        interface Timer<TMilli> as BeaconTimer;
        interface Timer<TMilli> as RouteTimer;
        interface Random;
        interface CollectionDebug;
        interface CtpCongestion;
    }
}

interface UnicastNameFreeRouting {
  command am_addr_t nextHop();

  command bool hasRoute();
  event void routeFound();
  event void noRoute();
}
```

4.3 CtpForwardingEngineP

The CtpForwardingEngineP component provides all the top level interfaces (except RootControl) which CollectionC provides and an application uses. It deals with retransmissions, duplicate suppression, packet timing, loop detection, and also informs the LinkEstimator of the outcome of attempted transmissions.:

```
generic module CtpForwardingEngineP() {
  provides {
    interface Init;
    interface StdControl;
    interface Send[uint8_t client];
    interface Receive[collection_id_t id];
    interface Receive as Snoop[collection_id_t id];
    interface Intercept[collection_id_t id];
    interface Packet;
    interface CollectionPacket;
    interface CtpPacket;
    interface CtpCongestion;
  }
  uses {
    interface SplitControl as RadioControl;
    interface AMSend as SubSend;
    interface Receive as SubReceive;
    interface Receive as SubSnoop;
    interface Packet as SubPacket;
    interface UnicastNameFreeRouting;
    interface Queue<fe_queue_entry_t*> as SendQueue;
    interface Pool<fe_queue_entry_t> as QEntryPool;
    interface Pool<message_t> as MessagePool;
    interface Timer<TMilli> as RetxmitTimer;
    interface LinkEstimator;
    interface Timer<TMilli> as CongestionTimer;
    interface Cache<message_t*> as SentCache;
    interface CtpInfo;
    interface PacketAcknowledgements;
    interface Random;
    interface RootControl;
    interface CollectionId[uint8_t client];
    interface AMPacket;
    interface CollectionDebug;
  }
}
```

CtpForwardingEngineP uses a large number of interfaces, which can be broken up into a few groups of functionality:

- Single hop communication: SubSend, SubReceive, SubSnoop, SubPacket, PacketAcknowledgments, AMPacket
- Routing: UnicastNameFreeRouting, RootControl, CtpInfo, CollectionId, SentCache
- Queue and buffer management: SendQueue, MessagePool, QEntryPool
- Packet timing: Random, RetxmitTimer

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)