Dissemination

TEP:	118
Group:	Net2 Working Group
Type:	Documentary
Status:	Draft
TinyOS-Version:	2.x
Author:	Philip Levis and Gilman Tolle
Draft-Created:	10-Dec-2004
Draft-Version:	1.7
Draft-Modified:	2007-06-14
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 for disseminating small (smaller than a single packet payload) pieces of data in TinyOS 2.x. Dissemination is reliably delivering a piece of data to every node in a network.

1. Introduction

Dissemination is a basic sensor network protocol. The ability to reliably deliver a piece of data to every node allows administrators to reconfigure, query, and reprogram a network. Reliability is important because it makes the operation robust to temporary disconnections or high packet loss. Unlike flooding protocols, which are discrete efforts that terminate, possibly not delivering the data to some nodes, dissemination achieves reliability by using a continuous approach that can detect when a node is missing the data.

Depending on the size of the data item, dissemination protocols can differ greatly: efficiently disseminating tens of kilobytes of a binary requires a different protocol than disseminating a two-byte configuration constant. Looking more deeply, however, there are similarities. Separating a dissemination protocol into two parts --- control traffic and data traffic --- shows that while the data traffic protocols are greatly dependent on the size of the data item, the control traffic tends to be the same or very similar.

Being able to reliably disseminate small values into a network is a useful building block for sensornet applications. It allows an administrator to inject small programs or commands and configuration constants. Because TinyOS nodes have limited RAM, these dissemination services have the assumption

that data values have some form of versioning. Dissemination propagates only the most recent version. This means that if a node is disconnected from a network and the network goes through eight versions of a disseminated value, when it rejoins the network it will only see the most recent. The rest of this document describes a set of components and interfaces for a dissemination service of this kind.

2. Dissemination interfaces

Small-value dissemination has two interfaces: DisseminationValue and DisseminationUpdate. The former is for consumers of a disseminated value, the latter is for producers. They are as follows:

```
interface DisseminationValue<t> {
  command const t* get();
  event void changed();
}

interface DisseminationUpdate<t> {
  command void change(t* newVal);
}
```

These interfaces assume that the allocation for the disseminated data is within the dissemination service. A consumer can obtain a const pointer to the data through DissemnationValue.get(). It MUST NOT store this pointer, as it may not be constant across updates. Additionally, doing so wastes RAM, as it can be easily re-obtained. The service signals a changed() event whenever the dissemination value changes, in case the consumer needs to perform some computation on it.

DisseminationUpdate has a single command, `change`, which takes a pointer as an argument. This pointer is not stored: a provider of DisseminationUpdate MUST copy the data into its own allocated memory.

A dissemination protocol MUST reach consensus on the newest value in a network (assuming the network is connected). Calling change implicitly makes the data item "newer" so that it will be disseminated to every node in the network. This change is local, however. If a node that is out-of-date also calls change, the new value might not disseminate, as other nodes might already have a newer value. If two nodes call change at the same time but pass different values, then the network might reach consensus when nodes have different values. The dissemination protocol therefore MUST have a tie-breaking mechanism, so that eventually every node has the same data value.

3 Dissemination Service

A dissemination service MUST provide one component, DisseminatorC, which has the following signature:

```
generic configuration DisseminatorC(typedef t, uint16_t key) {
  provides interface DisseminationValue <t>;
  provides interface DisseminationUpdate <t>;
}
```

The t argument MUST be able to fit in a single message_t [TEP111] after considering the headers that the dissemination protocol introduces. A dissemination implementation SHOULD have a compile error if a larger type than this is used.

As each instantiation of DisseminatorC probably allocates storage and generates code, if more than one component wants to share a disseminated value then they SHOULD encapsulate the value in a non-generic component that can be shared. E.g.:

```
configuration DisseminateTxPowerC {
```

```
  provides interface DisseminationValue<uint8_t>;
}
implementation {
  components new DisseminatorC(uint8_t, DIS_TX_POWER);
  DisseminationValue = DisseminatorC;
}
```

Two different instances of DisseminatorC MUST NOT share the same value for the `key` argument.

4 Dissemination Keys

One issue that comes up when using this interfaces is the selection of a key for each value. On one hand, using unique() is easy, but this means that the keyspaces for two different compilations of the same program might be different and there's no way to support a network with more than one binary. On the other, having a component declare its own key internally means that you can run into key collisions that can't be resolved. In the middle, an application can select keys on behalf of other components.

Ordinarily, dissemination keys can be generated by unique or selected by hand. However, these defined keys can be overridden by an application-specific header file. The unique namespace and the static namespace are separated by their most significant bit. A component author might write something like this:

```
#include <disseminate_keys.h>
configuration SomeComponentC {
  ...
}
implementation {
#ifndef DIS_SOME_COMPONENT_KEY
  enum {
    DIS_SOME_COMPONENT_KEY = unique(DISSEMINATE_KEY) + 1 << 15;
  };
#endif
  components SomeComponentP;
  components new DisseminatorC(uint8_t, DIS_SOME_COMPONENT_KEY);
  SomeComponentP.ConfigVal -> DisseminatorC;
}
```

To override, you can then make a disseminate_keys.h in your app directory:

```
#define DIS_SOME_COMPONENT_KEY 32
```

Even with careful key selection, two incompatible binaries with keyspace collisions may end up in the same network. If this happens, a GUID that's unique to a particular binary MAY be included in the protocol. The GUID enables nodes to detect versions from other binaries and not store them. This GUID won't be part of the external interface, but will be used internally.

5. More Complex Dissemination

An application can use this low-level networking primitive to build more complex dissemination systems. For example, if you want have a dissemination that only nodes which satisfy a predicate receive, you can do that by making the <t> a struct that stores a predicate and data value in it, and layering the predicate evaluation on top of the above interfaces.

6. Implementation

An implementation of this TEP can be found in `tinyos-2.x/tos/lib/net`. This dissemination implementation uses network trickles[2]. Each dissemination value has a separate trickle.

6. Author's Address

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

phone - +1 650 725 9046

Gilman Tolle
2168 Shattuck Ave.
Arched Rock Corporation
Berkeley, CA 94704

phone - +1 510 981 8714
email - gtolle@archedrock.com

7. Citations

[1] TEP 111: message_t.

[2] 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).