Dissemination of Small Values

TEP:	118
Group:	Net2 Working Group
Type:	Documentary
Status:	Final
TinyOS-Version:	2.x
Author:	Philip Levis and Gilman Tolle

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 establishes eventual consistency across the entire network on a shared variable and tells an application when the variable changes. Common uses of this mechanism include network reconfiguration and reprogramming.

1. Introduction

Dissemination is a service for establishing eventual consistency on a shared variable. Every node in the network stores a copy of this variable. The dissemination service tells nodes when the value changes, and exchanges packets so it will reach eventual consistency across the network. At any given time, two nodes may disagree, but over time the number of disagreements will shrink and the network will converge on a single value.

Eventual consistency is robust to temporary disconnections or high packet loss. Unlike flooding protocols, which are discrete efforts that terminate and not reach consistency, dissemination assures that the network will reach consensus on the value as long as it is not disconnected.

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. For example, the Deluge binary reprogramming service disseminates metadata about the

binaries. When nodes learn the disseminated metadata differs from the metadata of their local binary, they know they either have a bad binary or need a new one.

Novelty is an explicit consideration in dissemination's consistency model: it seeks to have every node agree on the most recent version of the variable. In this way, a node can prompt the network to reach consistency on a new value for a variable by telling the network it is newer. If several nodes all decide to update the variable, dissemination ensures that the network converges on a single one of the updates.

Consistency does not mean that every node will see every possible value the variable takes: it only means that the network will eventually agree on what the newest is. If a node is disconnected from a network and the network goes through eight updates to a shared variable, when it rejoins the network it will only see the most recent.

Being able to disseminate small values into a network is a useful building block for sensornet applications. It allows an administrator to inject small programs, commands, and configuration constants. For example, installing a small program through the entire network can be cast as the problem of establishing consistency on a variable that contains the program.

The rest of this document describes a set of components and interfaces for a dissemination service included in TinyOS 2.0. This service only handles small values that can fit in a single packet. Larger values require different interfaces and abstractions.

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();
  command void set(const t*);
  event void changed();
}

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

These interfaces assume that the dissemination service allocates space to store the variable. In that way, multiple components can all access and share the same variable that the dissemination service establishes consistency over. 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 or take some action.

DisseminationValue has a command, "set", which allows a node to change its local copy of a value without establishing consistency. This command exists so a node can establish an initial value for the variable. A node MUST NOT call "set" after it has handled a "changed" event, or the network may become inconsistent. If a node has received an update or a client has called "change" then set MUST NOT apply its new value.

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. DisseminationValue MUST signal ”changed” in response to a call to ”change”.

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 these 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 hand, 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 causes only nodes which satisfy a predicate to apply a change, 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

Two implementations of this TEP exist and can be found in `tinyos-2.x/tos/lib/net`. The first, Drip, can be found in "tinyos-2.x/tos/lib/net/drip". The second, DIP, can be found in "tinyos-2.x/tos/lib/net/dip". Both implementations are based on the Trickle algorithm[2]_. Drip is a simple, basic implementation that establishes an independent trickle for each variable. DIP uses a more complex approach involving hash trees, such that it is faster, especially when the dissemination service is maintaining many variables[3]_. This complexity, however, causes DIP to use more program memory.

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).

[3] Kaisen Lin and Philip Levis. "Data Discovery and Dissemination with DIP." In Proceedings of the Proceedings of the Seventh International Conference on Information Processing in Wireless Sensor Networks (IPSN), 2008.