Virtualization

TEP:	110
Group:	Core Working Group
Type:	Documentary
Status:	Draft
TinyOS-Version:	2.x
Author:	Philip Levis
Draft-Created:	20-Jun-2005
Draft-Version:	1.5
Draft-Modified:	2006-12-12
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

This memo desribes how TinyOS 2.0 virtualizes common abstractions through a combination of static allocation and runtime arbitration. It describes the benefits and tradeoffs of this approach and how it is used in several major abstractions.

1. Introduction

The TinyOS component model allows flexible composition, but that flexibility is often limited by reasons which are not explicitly stated in components. These implicit assumptions can manifest as buggy behavior. In TinyOS 1.x, on the Telos platform, if a program simultaneously initializes non-volatile storage and the radio, one of them will fail: a program has to initialize them serially. They can also manifest as compile-time errors: if two separate communication services happen to receive packets of the same AM type, the nesC compiler will issue a warning due to the buffer swap semantics.

On one hand, the flexbility components provide allows expert users to build complex and intricate applications. On the other, it can make managing complexity and intricacy very difficult when building very simple and basic applications. To promote this latter class of development, TinyOS 2.x has the notion of "distributions," which are collections of system abstractions that are designed to be used together. As long as a user implements an application in terms of the distribution, the underlying components will operate correctly and there will be no unforseen failures.

This memo documents an example distribution, named OSKI (Operating System Key Interfaces). It describes the services OSKI provides and how their implementations are structured to simplify application writing.

2. Distributions

A distribution presents *services* to the programmer. A service is a set of generic (instantiable) components that represent API abstractions. To use an abstraction, a programmer instantiates the generic. For example, OSKI has the `AMSenderC` abstraction for sending active messages. The AMSender generic component takes a single parameter, the AM type. For example, if a programmer wants a component named AppM to send active messages of type 32, the configuration would look something like this:

```
configuration AppC {}
implementation {
    components AppM, new AMSenderC(32);
    AppM.AMSend -> AMSenderC;
}
```

Services often present abstractions at a fine grain. For example, the active message service has AMSender, AMReceiver, and AMSnooper, each of which is a separate abstraction.

2.1 Controlling a Service

Every service has two abstractions: `ServiceC`, for powering it on and off, and `ServiceNotifierC`, for learning when the service's power state has changed. For example, active messages have the `AMServiceC` and `AMServiceNotifierC` abstractions. A service abstraction provides the `Service` interface

```
interface Service {
    command void start();
    command void stop();
    command bool isRunning();
}
```

while a notifier abstraction provides the `ServiceNotify` interface

```
interface ServiceNotify {
    event void started();
    event void stopped();
}
```

For example, if a routing layer wants to be able to turn the active message layer on and off, then it needs to instantiate an AMServiceC. However, many components may be using active messages and have their own instances of AMServiceC; while routing might consider it acceptable to turn off active messages, other components might not.

Therefore, a service abstraction does not necessarily represent explicit control; instead, each service has a *policy* for how it deals with control requests. When a service changes its activity state, it MUST signal all instances of its ServiceNotifierC.

For example, the active messages service has an "OR" policy; the service remains active if *any* of its ServiceC instances are in the on state, and goes inactive if and only if *all* of its ServiceC instances are in the off state. This is an example timeline for active messages being used by two components, RouterA and RouterB:

1. System boots: active messages are off.
2. RouterA calls `Service.start()`. The AM layer is turned on.
3. All AMServiceNotifierC abstractions signal `ServiceNotify.started()`.
4. RouterB calls `Service.start()`.
5. RouterA calls `Service.stop()`. RouterB is still using active messages, so the layer stays on.
6. RouterB calls `Service.stop()`. The AM layer is turned off.
7. All AMServiceNotifierC abstractions signal `ServiceNotify.stopped()`.

By default, a service that has a control interface MUST be off. For an application to use the service, at least one component has to call `Service.start()`.

2.2 Service Initialization

Because distributions are collections of services that are designed to work together, they can avoid many of the common issues that arise when composing TinyOS programs. For example, user code does not have to initialize a service; this is done automatically by the distribution. If a user component instantiates a service abstraction, the distribution MUST make sure that the service is properly initialized. Section 4 goes into an example implementation of how a distribution can achieve this.

3. OSKI Services

This section briefly describes the services that OSKI, an example distribution provides. It is intended to give a feel for how a distribution presents its abstractions.

3.1 Timers

OSKI provides timers at one fidelity: milliseconds. Timers do not have a Service abstraction, as their use implicitly defines whether the service is active or not (the timer service is off if there are no pending timers). The `OSKITimerMsC` component provides the abstraction: it provides a single interface, `Timer<TMilli>`.

This is an example code snippet for instantiating a timer in a configuration:

```
configuration ExampleC {
    uses interface Timer<TMilli>;
}
```

```
configuration TimerExample {}
implementation {
    components ExampleC, new OSKITimerMsC() as T;
    ExampleC.Timer -> T;
}
```

3.2 Active Messages

OSKI provides four functional active messaging abstractions: `AMSender`, `AMReceiver`, `AMSnooper`, and `AMSnoopingReceiver`. Each one takes an `am_id_t` as a parameter, indicating the AM type. Following the general TinyOS 2.x approach to networking, all active message abstractions provide the `Packet` and `AMPacket` interfaces.

AMSender This abstraction is for sending active messages. In addition to Packet and AMPacket, it provides the AMSend interface.

AMReceiver This abstraction is for receiving active messages addressed to the node or to the broadcast address. In addition to Packet and AMPacket, it provides the Receive interface.

AMSnooper This abstraction is for receiving active messages addressed to other nodes ("snooping" on traffic). In addition to Packet and AMPacket, it provides the Receive interface.

AMSnoopingReceiver A union of the functionality of AMReceiver and AMSnooper, this abstraction allows a component to receive *all* active messages that it hears. The AMPacket interface allows a component to determine whether such a message is destined for it. In addition to Packet and AMPacket, this component provides the Receive interface.

This snippet of code is an example of a configuration that composes a routing layer with needed active message abstractions. This implementation snoops on data packets sent by other nodes to improve its topology formation:

```
configuration RouterC {
    uses interface AMSend as DataSend;
    uses interface AMSend as ControlSend;
    uses interface Receive as DataReceive;
    uses interface Receive as BeaconReceive;
}

configuration RoutingExample {
    components RouterC;
    components new AMSender(5) as CSender;
    components new AMSender(6) as DSender;
    components new AMReceiver(5) as CReceiver;
    components new AMSnoopingReceiver(6) as DReceiver;
```

```
    RouterC.DataSend -> DSender;
    RouterC.ControlSend -> CSender;
    RouterC.DataReceive -> DReceiver;
    RouterC.ControlReceive -> CReceiver;
}
```

The active messages layer has control abstractions, named `AMServiceC` and `AMServiceNotifierC`. Active messages follow an OR policy.

3.3 Broadcasts

In addition to active messages, OSKI provides a broadcasting service. Unlike active messages, which are addressed in terms of AM addresses, broadcasts are address-free. Broadcast communication has two abstractions: `BroadcastSenderC` and `BroadcastReceiverC`, both of which take a parameter, a broadcast message type. This parameter is similar to the AM type in active messages. Both abstractions provide the Packet interface. The broadcast service has control abstractions, named `BroadcastServiceC` and `BroadcastServiceNotifierC`, which follow an OR policy.

3.4 Tree Collection/Convergecast

NOTE: These services are not supported as of the 2.x prerelease. They will be supported by the first full release.

OSKI's third communication service is tree-based collection routing. This service has four abstractions:

CollectionSenderC This abstraction is for sending packets up the collection tree, to the collection root. It provides the Send and Packet interfaces.

CollectionReceiverC This abstraction is for a collection end-point (a tree root). It provides the Receive, Packet, and CollectionPacket interfaces.

CollectionInterceptorC This abstraction represents a node's ability to view and possibly suppress packets it has received for forwarding. It provides the Intercept, CollectionPacket, and Packet interfaces.

CollectionSnooperC This abstraction allows a node to overhear routing packets sent to other nodes to forward. It provides the Receive, CollectionPacket, and Packet interfaces.

All of the collection routing communication abstractions take a parameter, similar to active messages and broadcasts, so multiple components can independelty use collection routing. In addition to communication, collection routing has an additional abstraction:

CollectionControlC This abstraction controls whether a node is a collection root or not.

Finally, collection routing has `CollectionServiceC` and `CollectionServiceNotifierC` abstractions, which follow an OR policy.

3.5 UART

NOTE: These services are not supported as of the 2.x prerelease. They will be supported by the first full release. They will be fully defined pending discussion/codification of UART interfaces.

4. OSKI Service Structure and Design

Presenting services through abstractions hides the underlying wiring details and gives a distribution a great deal of implementation freedom. One issue that arises, however, is initialization. If a user component instantiates a service, then a distribution MUST make sure the service is initialized properly. OSKI achieves this by encapsulating a complete service as a working component; abstractions export the service's interfaces.

4.1 Example: Timers

For example, the timer service provides a single abstraction, OskiTimerMilliC, which is a generic component. OskiTimerMilliC provides a single instance of the Timer<TMilli> interface. It is implemented as a wrapper around the underlying timer service, a component named `TimerMilliImplP`, which provides a parameterized interface and follows the Service Instance design pattern[sipattern]_:

```
generic configuration OskiTimerMilliC() {
    provides interface Timer<TMilli>;
}
implementation {
    components TimerMilliImplP;
    Timer = TimerMilliImplP.TimerMilli[unique("TimerMilliC.TimerMilli")];
}
```

TimerMilliImplP is a fully composed and working service. It takes a platform's timer implementation and makes sure it is initialized through the TinyOS boot sequence[boot]_:

```
configuration TimerMilliImplP {
    provides interface Timer<TMilli> as TimerMilli[uint8_t id];
}
implementation {
    components TimerMilliC, Main;
    Main.SoftwareInit -> TimerMilliC;
    TimerMilli = TimerMilliC;
}
```

This composition means that if any component instantiates a timer, then TimerMilliImplP will be included in the component graph. If TimerMilliImplP is included, the TimerMilliP (the actual platform HIL implementation) will be properly initialized at system boot time. In this case, the order of initialization isn't important; in cases where

there are services that have to be initialized in a particular sequence to ensure proper ordering, the Impl components can orchestrate that order. For example, a distribution can wire Main.SoftwareInit to a DistributionInit component, which calls sub-Inits in a certain order; when a service is included, it wires itself to one of the sub-Inits.

The user does not have to worry about unique strings to manage the underlying Service Instance pattern: the abstractions take care of that.

4.2 Example: Active Messages

Active messaging reprsent a slightly more complex service, as it has several abstractions and a control interface. However, it follows the same basic pattern: abstractions are generics that export wirings to the underlying service, named `ActiveMessageImplP`:

```
configuration ActiveMessageImplP {
    provides {
        interface SplitControl;
        interface AMSend[am_id_t id];
        interface Receive[am_id_t id];
        interface Receive as Snoop[am_id_t id];
        interface Packet;
        interface AMPacket;
    }
}

implementation {
    components ActiveMessageC, Main;

    Main.SoftwareInit -> ActiveMessageC;

    SplitControl = ActiveMessageC;
    AMSend = ActiveMessageC;
    Receive = ActiveMessageC.Receive;
    Snoop = ActiveMessageC.Snoop;
    Packet = ActiveMessageC;
    AMPacket = ActiveMessageC;
}
```

For example, this is the AMSender abstraction:

```
generic configuration AMSenderC(am_id_t AMId) {
    provides {
        interface AMSend;
        interface Packet;
        interface AMPacket;
    }
```

```
}

implementation {
    components ActiveMessageImplP as Impl;

    AMSend = Impl.AMSend[AMId];
    Packet = Impl;
    AMPacket = Impl;
}
```

AMReceiver is similar, except that it wires to the Receive interface, while AMSnooper wires to the Snoop interface, and AMSnoopingReceiver provides a single Receive that exports both Snoop and Receive (the unidirectional nature of the Receive interface makes this simple to achieve, as it represents only fan-in and no fan-out).

ActiveMessageImplP does not provide a Service interface; it provides the SplitControl interface of the underlying active message layer. OSKI layers a *ServiceController* on top of SplitControl. As the active message service follows an OR policy, OSKI uses a `ServiceOrControllerM`, which is a generic component with the following signature:

```
generic module ServiceOrControllerM(char strID[]) {
    provides {
        interface Service[uint8_t id];
        interface ServiceNotify;
    }
    uses {
        interface SplitControl;
    }
}
```

ServiceOrControllerM follows the Service Instance pattern[sipattern]; it calls its underlying SplitControl based on the state of each of its instances of the Service interface. The parameter denotes the string used to generate the unique service IDs. The active messages service controller implementation, AMServiceImplP, instantiates a ServiceOrControllerM, wires it to ActiveMessageImplP:

```
configuration AMServiceImplP {
    provides interface Service[uint8_t id];
    provides interface ServiceNotify;
}
implementation {
    components ActiveMessageImplP;
    components new ServiceOrControllerM("AMServiceImplP.Service");

    Service = ServiceOrControllerM;
    ServiceOrControllerM.SplitControl -> ActiveMessageImplP;
```

```
    ServiceNotify = ServiceOrControllerM;
}
```

AMServiceC then provides an instance of AMServiceImplP.Service:

```
generic configuration AMServiceC() {
    provides interface Service;
}

implementation {
    components AMServiceImplP;

    Service = AMServiceImplP.Service[unique("AMServiceImplP.Service")];
}
```

Note that the two strings are the same, so that the uniqueCount() in the ServiceOrControllerM is correct based on the number of instances of AMServiceC. As with timers, encapsulating the service instance pattern in generic components relieves the programmer of having to deal with unique strings, a common source of bugs in TinyOS 1.x code.

4.3 OSKI Requirements

OSKI is a layer on top of system components: it presents a more usable, less error-prone, and simpler interface to common TinyOS functionality. For OSKI to work properly, a platform MUST be compliant with the following TEPs:

o TEP 102: Timers o TEP 106: Schedulers and Tasks o TEP 107: TinyOS 2.x Boot Sequence o TEP 1XX: Active Messages o TEP 1XX: Collection Routing

Not following some of these TEPS MAY lead to OSKI services being inoperable, exhibit strange behavior, or being uncompilable.

5. Distribution Interfaces

The basic notion of a distribution is that it provides a self-contained, tested, and complete (for an application domain) programming interface to TinyOS. Layers can be added on top of a distribution, but as a distribution is a self-contained set of abstractions, adding new services can lead to failures. A distribution represents a hard line above which all other code operates. One SHOULD NOT add new services, as they can disrupt the underlying organization. Of course, one MAY create a new distribution that extends an existing one, but this is in and of itself a new distribution.

Generally, as distributions are intended to be higher-level abstractions, they SHOULD NOT provide any asynchronous (async) events. They can, of course, provide async commands. The idea is that no code written on top of a distribution should be asynchronous, due to the complexity introduced by having to manage concurrency. Distributions are usually platform independent; if an application needs async events, then chances are it is operating close to the hardware, and so is not platform independent.

6. Author's Address

Philip Levis
467 Soda Hall
UC Berkeley
Berkeley, CA 94720

phone - +1 510 290 5283

email - pal@cs.berkeley.edu

7. Citations

[rst] reStructuredText Markup Specification. <http://docutils.sourceforge.net/docs/ref/rst/restructuredtext.html>

[sipattern] The Service Instance Pattern. In *Software Design Patterns for TinyOS.* David Gay, Philip Levis, and David Culler. Published in Proceedings of the ACM SIGPLAN/SIGBED 2005 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'05).

[boot] TEP 107: TinyOS 2.x Boot Sequence.