NestArch: Prototype Time Synchronization Service

Ted Herman, University of Iowa, herman@cs.uiowa.edu

6 January 2003

Abstract

This document is a brief explanation of a rudimentary time service (Tsync) developed for the NEST Challenge Architecture. The document describes the basic design, performance expectations for Tsync, and its TinyOS implementation.

1 Introduction

Time synchronization in a distributed, real-time, system is a basic service. It can be useful for the NEST Challenge Project in several ways: calculation of velocities, coordination of middleware services, and resource allocation algorithms. Initial specifications for a time synchronization service were proposed in the Fall of 2002; subsequent experience implementing time synchronization in the mote architecture, along with the pressure of project deadlines, have motivated a few modifications to the initial specification. However the spirit of the original specification remains in place: global system time is represented by an integer, denominated in 30.5176 μsec units (*ie*, 32,768 ticks per second). The remainder of this document explains the basics of the initial implementation, consequences of its simple design with respect to expected performance, and avenues for improvement. Throughout this document we refer to the time synchronization service as Tsync.

2 The Interface

Users of Tsync will be happy to see that it offers a simple interface. In addition to the standard control interface `StdControl` (used by many components), Tsync supports the `Time` interface. This interface provides two commands for obtaining the current time, `getGlobalTime` and `getLocalTime`. The caller of these commands supplies a pointer to a `timeSync` structure, which is nothing more than a `uint32`-integer to store the time (in future, this structure could become

richer). In most practice, there is no difference between these two commands; during initialization of a mote network, `getGlobalTime` could return a `FAIL` status in case mote clocks have not yet been synchronized, whereas `getLocalTime` always succeeds.

In the initial draft of the time synchronization time was specified as a `uint48`-integer; elsewhere, Su Ping proposed that time be represented by `uint64`-integers. We chose `uint32` for the present because it is adequate for the challenge demonstration and can easily be changed later. Note that `uint32`-valued time corresponds to about 36 hours.

Earlier in our efforts to specify a time synchronization service, we considered the possibility of regional time bases (local time, somewhat like a "time zone") such as the Reference Broadcast System [1] uses. Other possibilities not contemplated include using time intervals (a pair of numbers denotes an interval, and this interval contains the true time), vectors or matrices, and even graph-theoretic structures. Also, it is reasonable to consider enhancing the `timeSync` structure someday to have indications of the quality of the clock returned by a call to the service. In addition, the interface could have interactions for long-term energy saving, tuning, and other factors.

3 Some Facts about Mote Clocks

We explain some of the basics about how motes deal with clocks, since this influences our implementation and explains some of the performance consequences.

Motes do not have "clocks" as such; rather, some registers of the processor can be programmed (subject to several limitations) to increment with each processor cycle (*eg* at 4 MHz) and generate an interrupt upon reaching a maximum count. Therefore we've got to simulate a clock. Since our desired clock precision is 32768 Hz, our simulated clock should ideally advance once per 122.07 processor cycles. This turns out to be impractical for two reasons. First, the effort of processing interrupts to increment a software counter is too expensive (competing with other processing needs); second, the processor is not so flexible in how it can program its hardware counters.

The hardware counter behind the simulated clock can be programming in eight different granularities, and for each of these granularities there is a multiplier. If the granularity is set at 32768 Hz, then an interrupt will occur after 30.5176 μsec (minimum) up to 7.78198 msec (maximum), depending on the value of the 8-bit multiplier. Other granularities are 4096, 1024, 512, 256, 128, 32, and 0 Hz. This implies that if we desire an interrupt to occur about every second, either a multiplier of 32 with a granularity of 32 can be used or a multiplier of 128 with a granularity

of 128 can be used; higher granularities won't work if we want one clock interrupt per second. Why is this important? It has consequences for our desired implementation of a time service with 32 KHz precision. It is just too expensive to actually increment the clock at 32768 Hz; however if the hardware counter is programmed to interrupt once per second, then each interrupt will trigger

```
clock = clock + 32768
```

the overhead is acceptably low, and we have the desired result!? Well, not really. What if some application queries the time service for the current clock *between* the counter-driven interrupts? The accuracy will be around half a second on average – we might as well use a clock whose units are half seconds. Fortunately, the story doesn't end here.

The processor can also *read* the value of the hardware counter, before it has generated an interrupt. Thus an instantaneous reading of time is possible. We implemented this approach for Tsync. Its accuracy is, however, dependent on the granularity of the hardware counter. If the timer has been set to fire once per second, then it sets the granularity to 128 Hz, so an instantaneous reading will have an accuracy of about 3.9 milliseconds on average (and each counter-driven interrupt adds 256 to the counter). We anticipate this to be the norm for timer settings (firing once per second). When the timer is set to higher frequency, say 10 millisecond firing, then the granularity does become 32768 Hz, but this is expensive and only to be used sparingly. Applications that need to measure the difference between two times obtained from Tsync therefore should be designed in the context of timer (*ie* the TinyOS `Timer` component) settings.

In developing the prototype Tsync implementation, we calibrated its software clock using a GPS-delivered pulse-per-second signal. We observed that Tsync time is about 1.5% slower than real time at typical timer settings; at the highest granularity, the clock ran about 3% slower than real time. We speculate that most of this drift is due to delays of interrupt processing. In the initial version, there is no correction to the software clock to compensate for this observed phenomenon.

4 Basic Design

The constraints for our basic design are: (1) simplicity; (2) some fault tolerance; (3) no built-in dependence on base stations or specialized mote roles.

Our design implicitly elects the mote with the smallest identifier to be the "root clock". The root clock periodically announces the value of its Tsync-time; other motes copy the root clock. Thus this is a *push* design (whereas NTP uses a pull

method to synchronize). Only one message type (`Beacon`) is used by the `Tsync` component.

Each non-root mode chooses, among its set of neighbors, one with least distance (measured in hops) to the root. The neighbor chosen supplies its clock as an approximation to the root time. The set of neighbors is determined by building a table based on received beacon messages. Beacon messages contain fields to name the root identifier, the number of hops to the root, the sender's clock at the instant of sending, and a few other values. In the initial implementation, there is no correction for the latency of sending and receiving a beacon message (we estimate the latency to be about 36 milliseconds).

What happens if a mote dies, or a link is somehow lost between motes? Tsync uses an aging technique to maintain the neighbor table. If a beacon hasn't been received for too long, then a neighbor is removed from the table. If, as a result, no neighbor can offer a path to the current root, a mote will adopt a new root. Technically this is done by the well-known "count to infinity" method of distance-vector routing protocols. The Tsync implementation uses a hard-coded limit on feasible hop distances to force the eventual extinguishing of path information. Thus if a root mote dies, eventually a new root emerges.

What happens if a new mote, which has a lower identifier than any previously known, enters the system? We didn't want the entire system to adopt the new mote's clock as it became root. Therefore Tsync's implementation refuses to accept the clock in a beacon message unless that beacon's sender has a nonempty neighborhood; and a mote with an empty neighborhood accepts the clock of the first beacon (with nonempty neighborhood) that it receives.

To be sure, there are many other failure cases that Tsync does not cover; better fault tolerance is just one of the areas of improvement for future versions of Tsync.

5 Performance

As the reader can see from the basic design, Tsync doesn't offer anything in the way of performance guarantees. We guess that clocks will be within about $36 \cdot h$ milliseconds of the root mote's, where h is the number of hops to the root. The clocks are adjusted whenever a new root is detected or whenever some neighbor has a smaller clock than the current time. Thus Tsync's clocks are *not monotonic* in the initial version.

References

[1] J Elson, L Girod, D Estrin. Fine-grained network time synchronization using reference broadcasts. May 2002.