EMBS Autumn Assessment

Y6385133

Autumn 2012

1 Modelling

I chose to model the solution using Mote Runner SDK. The main reasons against using another solution were:

Differences in implementation details
: There is no guarantee that applications such as ptollemy would exhibit the same characteristics as a mote

Time constraints
: I expected the majority of my time would be taken up with debugging the mote SDK

2 Implementation

My original [naive] implementation obvserved a channel until it had determined the period, at which point it analysed the next sink. This worked, but in the worst case situation where the first sink has $n = 1500$ and $t = 10$ the source could block for up to 35s – over half the time of the entire simulation – before it analyses other sinks. See figure 1

To get around this I added a timeout that restricts the amount of time the source can spend trying to analyse a specific channel when there are other periods that need estimating.

The general case for determining a sink's period involves observing two of the sequence numbers (n_1, n_2) emitted by a sink, recording the times they were observed (r_1, r_2) and then calculating:

$$t = \frac{r_2 - r_1}{n_1 - n_2}$$

This allows the period to be calculated without observing consecutive sequence numbers, however the two sequence numbers must be observed within the same synchronisation phase. In the case where $n = 1$ this is impossible, so I added a special case that calculates

$$t = \frac{r_2 - r_1}{12}$$

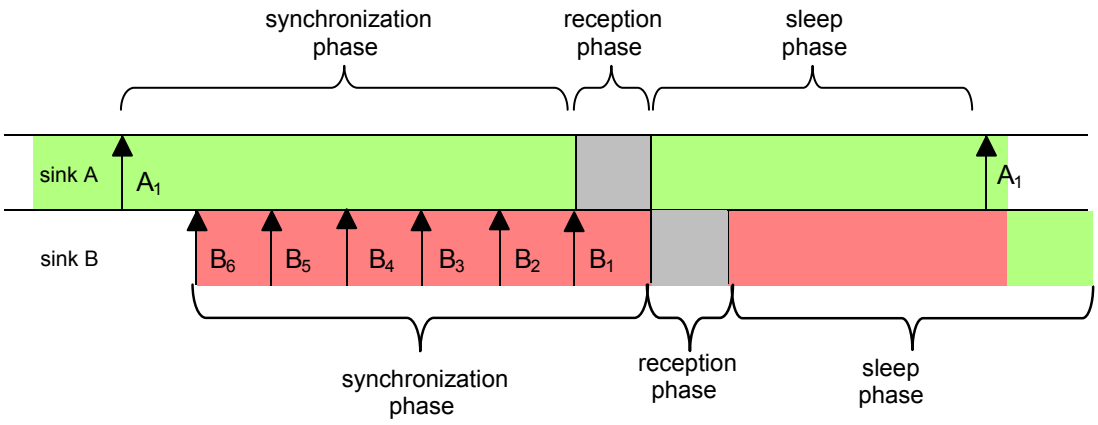


Figure 1: Blocking stuff

When $n_2 > n_1$ we cannot reliably calculate the period as the second sequence number may not be the sink's n_max, thus giving a larger t, which could cause us to transmit outside of the reception period. In this instance $n1$ is replaced with n_2's values and n_2 is reset.

3 Clock Drift

This is inevitable in systems without a centralised clock, especially when wireless communication is involved. To help combat drift the source always performs calculations using times events were triggered at, rather than the 'current' time. After each broadcast the source schedules a new broadcast in $t * (11 + n)$ ticks and a resync in $(t * (10 + n)) - x$ ticks. The resync is used to verify that the estimated reception period is accurate.

4 Power Saving

My implementation saves power by scaling down the transmission power based on the signal strength of received packets. Received power strength is between $0 - 255$ and transmission power $0 - 63$. Calculating min power needed to send to $sink_X$:

$$tx\_power_x = \frac{rx\_power_x}{255} \times 63$$

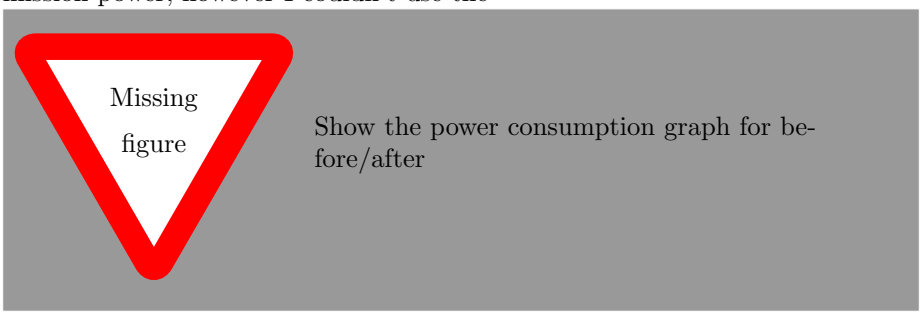
However using this exact value will mean a different min power is required as noise/distance from sink changes. To get around this I transmit at double the minimum required power.

$$tx\_power_x = max(63, \frac{rx\_power_x}{255} \times 63)$$

The above operation can be approximated using multiplications of powers of 2, aka bit shifts.

$$\begin{aligned} tx\_power_x &= rx\_power_x \times 2^{-8} \times 2^6 \\ &= rx\_power_x \times 2^{-2} \end{aligned}$$

The SDK's netview tool was used to confirm the reduction in signal transmission power, however I couldn't use the



5 Verification

Mote runner's dashboard, console and power-tracing tools were used to verify my solution.

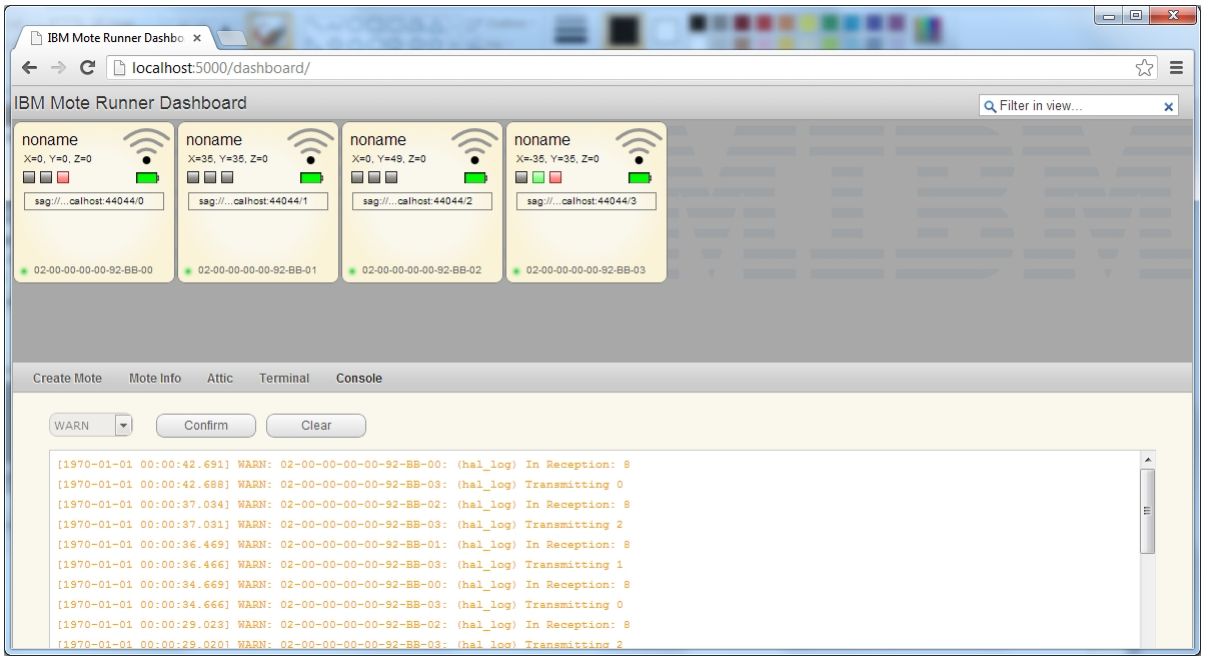


Figure 2: Moterunner Dashboard

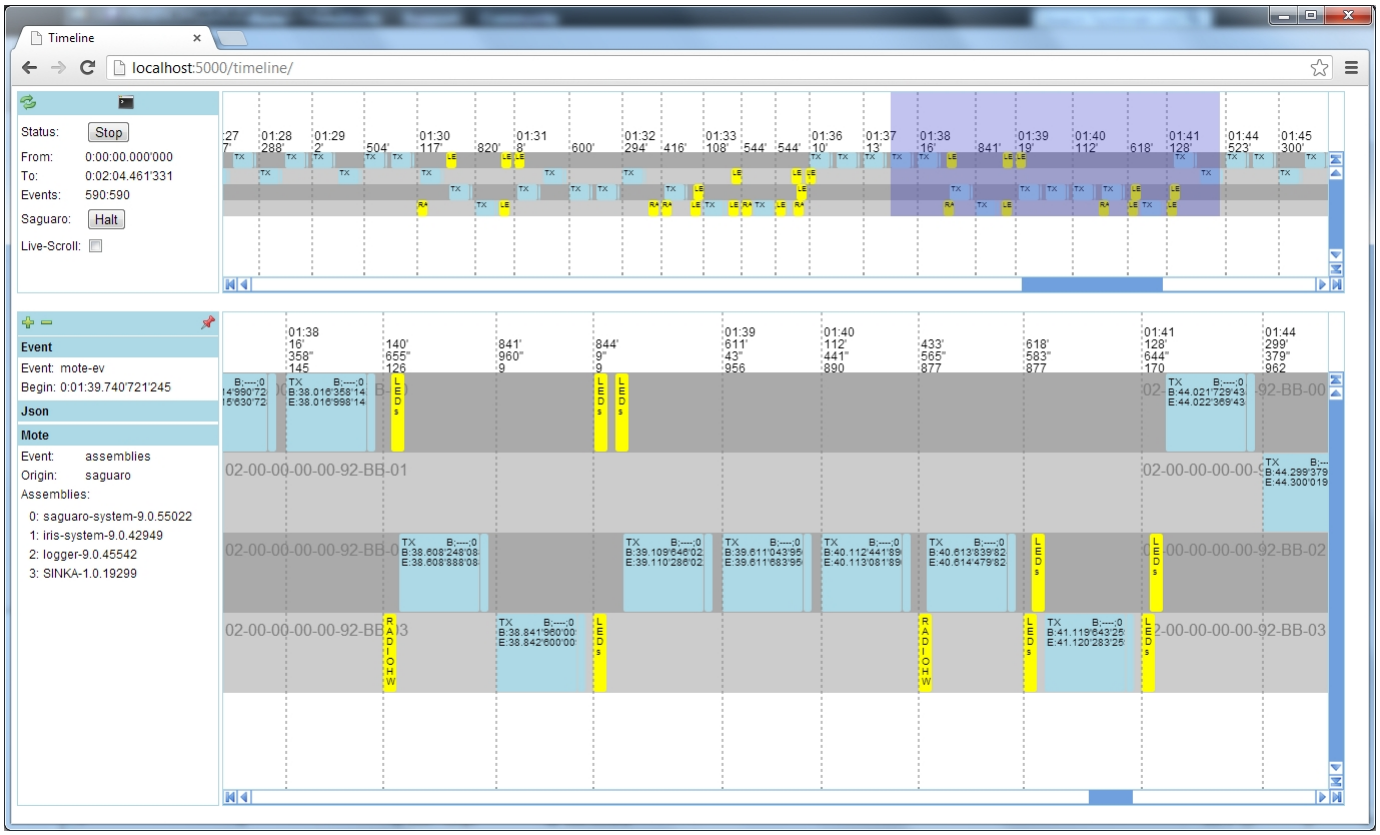


Figure 3: Timeline view The first three rows are sinks, the third is the source