Haywire: Bootstrap Letter

Table of contents

1 Foreword.......... 2
2 Better 1-Wire® API.......... 2
2.1 Mutual Exclusive Locking.......... 2
2.2 Seamless arrival/departure notifications.......... 3
2.3 Preserving the device state.......... 3
2.4 Raising the 1-Wire® abstraction level.......... 3
2.5 Automatic handling of the network path to the device.......... 4
3 Conclusion.......... 4

1. Foreword

Note:
Following is the document that was written to define the project scope. It will be kept here for historical purposes, so it may become obsolete sooner or later. However, it gives a pretty good idea of the itch that had to be scratched.

1-Wire® network is a technological wonder. It is a low cost, high reliability and high flexibility network that can run on as much as a regular telephone wire. There's quite a few hard working folks at Dallas Semiconductor that keep producing new devices and writing drivers for them, and the importance of their work can't be underestimated - all in all, they've made it possible and available for all of us to enjoy.

Having said that, let me try to express some constructive criticism and try to make the 1-Wire® devices even more attractive than they are today.

2. Better 1-Wire® API

The 1-Wire API done by Dallas Semiconductors was obviously developed from bottom up (the reason for that is very clear - the devices were introduced one or a few at a time, and the vision for the API was changing along the way). The interface is incredibly low-level, and you have to be quite a contortionist to achieve anything at all. Just the headlines: mutual exclusive locking, preserving the device state, seamless arrival/departure notifications, automatic handling of the network path to the device.

A user-oriented API is needed, with abstraction layer boundaries laid out as the user sees fit, not as hardware requires. This is quite an endeavour, though.

Let's expand the headlines a little bit.

2.1. Mutual Exclusive Locking

Currently, all the calls to time or sequence sensitive 1-Wire operations must be wrapped into `beginExclusive()/endExclusive()` pair. However, there's quite a few problems with the way it's done:

- (strategic) The lock ownership is anonymous. It is not possible to tell what entity currently has a lock on the adapter. Debugging a lost match between `beginExclusive()` and `endExclusive()` is a nightmare.
- (strategic) The locks are not enforced. You want to use the lock, go ahead and use it. You don't want to use it (or God forbid, you don't know about its existence) - nobody will prevent you from doing so, but you're in trouble already.

- (strategic) There's no way to prioritize the lock acquisition. In real life, there are things that may need to be done with different priority. For example, for fairly big but fairly stable networks the need to update the set of devices currently present on the network is not that critical, however, the rest of the operation may need to be carried out pretty fast. On the other hand, for the networks handling authorization using iButtons, the situation is the exact opposite: devices arriving on the network must be handled right away.
- (tactical) Waiting for the lock is done by means of repeatedly calling `Thread.sleep(50)`, as opposed to using synchronized access.

Bottomline: an enforceable mutual exclusive locking system supporting accountability and prioritization would be a nice addition to the API.

2.2. Seamless arrival/departure notifications

1-Wire® communication protocol seems to be able to handle that pretty good. However, as it stands now, the device arrival/departure must be monitored using a separate entity. To aggravate the situation, there's more than one - `OneWireMonitor` and `NetworkMonitor`, each of them having their own framework. Neither of them does the job completely - if the network topology goes beyond a simple bus (using DS2409 couplers), the full topology is not discovered properly. It would be nice to be able to receive an arrival/departure notification without consuming the 1-Wire® network nor host adapter resources at best, and definitely without extra hassle of having to keep track of the monitor.

2.3. Preserving the device state

As it stands now, a lot of 1-Wire® device state is preserved outside of the device abstraction, and the device abstracion is not coupled to the actual hardware device. For example, each call to `DSPortAdapter.getAllDeviceContainers()` will return a new device container instance each time, even though there could be instances controlling the given hardware devices already. You use a different instance, you lose the device state. This issue is tightly related to another - the abstraction level of the 1-Wire® device needs to be higher.

2.4. Raising the 1-Wire® abstraction level

Right now, the device container is nothing more than an adapter to the actual hardware device, preserving some of the device state. The state handling is inconsistent: in some cases it is preserved, in other cases it is not. Of course, doing RTFM and reading the code helps a lot, these inconsistencies can be discovered, but there's just too many of them. It would be nice to be able to treat the device container in exactly the same way as you treat the device.

To summarize last two paragraphs:

One hardware device, One container, One state, Indivisible.

Note:
See Design Contract for extension for this rule related to composite and group devices.

2.5. Automatic handling of the network path to the device

On a complex topology 1-Wire® network there are MicroLAN couplers. In order to get to the device somewhere at the branch, the corresponding chain, or path, of couplers has to be opened. However, current API doesn't provide abstractions to be able to discover a path to a specific 1-Wire® device except using the NetworkMonitor which is a) hassle to use b) doesn't cover all the cases. In particular, it is not able to discover the devices on the branches that are not currently opened.

Even if we get the path to the device right, it doesn't help much because there's an issue of opening that path every time the device needs to be accessed. There's a significant overhead related to opening a path - possibly multiple couplers need to change their state, plus, there's an overhead of making a decision about which exactly couplers have to be closed, opened, and whether they need to be touched at all (it is quite possible that the last accessed device was on the same path already).

This needs to be done for every application that deals with complex topology 1-Wire® networks, so it would be nice to have this provided behind the curtains.

3. Conclusion

The primary objective of the API discussed in this article is: simplicity. All the low-level details such as locking, device state preservation, path handling, network browsing and possibly more can be brought down below the level of abstraction the typical API user operates at.

Simplicity brings stability.

Stability brings confidence.

Confidence brings new users and new uses.

1-Wire® devices win.