Engineering Slow Technologies

Dr Tim Regan
Microsoft Research
Cambridge, UK
timregan@microsoft.com

Author Keywords
Slow technology, implementation, engineering.

ACM Classification Keywords
H5.m. Information interfaces and presentation (e.g., HCI): Miscellaneous.

INTRODUCTION

Looking back at the history of HCI we see a change in stance towards peoples' (users') interactions with the computer systems under our investigation. We started out concerned with efficiency and looked for predictive models to help us design fast and accurate user interactions. Then we concerned ourselves with the users' experiences and took a more holistic approach to designing experiences that were enjoyable and engaging. Now we turn our attention to meaningful interactions, meaningful in terms of human values. These include interactions that occur infrequently with days, weeks, months, or years between interactions. When engineering the systems that support these interactions we cannot rely on human interaction by users to troubleshoot potential problems. We need to take a new set of constraints into account.

Or do we?

This note, in submission to the DIS 2012 Workshop "Slow Technology" explores these engineering challenges through our (well my) experiences building a version of the Photobox system [1].

CONCERNS AND CHOICES

Monitoring

With a typical computer application, e.g. a web browser or a printer when it stops working you notice and take remedial action from restarting the application, power cycling the machine, through to calling out an engineer. Photobox was designed to sit unattended forgotten at the back of a cupboard and to only print out a photo inside the wooden box once in a blue moon, i.e. with a randomised frequency of four or so photos a year. This presents a monitoring challenge. Can we be sure that when the time comes the Photobox is alive and well enough to carry out the print run? During the months of inactivity has Bluetooth connectivity to the printer been lost? Have power or network outages rendered the Photobox useless? There are a number of potential solutions to this. Heart-beating, where the system periodically reports its status to the researcher running the experiment is one. Error reporting where self-diagnosed errors are reported to the experimenter is another. We chose a mixture of this and a third option whereby each Photobox will report its status when requested by the experimenter.

Robustness

For a system to remain live, without necessarily doing anything visible, over long periods of time many of the traditional robustness features of traditional software engineering like error handling and memory use issues need careful attention. Of course this is true of all systems, but those that will be periodically restarted, or those that are only ever going to be operated by the project member demoing the system have clearly lesser requirements for robustness. This may also dictate a choice for simpler implementations, or ever more complex fall-back measures. For example we use Twitter for Photoboxes to report their status and to read incoming instructions from the experimenter. This seemed an interesting decision at the time, allowing us to use our Photobox project to explore an aspect of 'The Internet of Things'. With hindsight though a simple web service implemented ourselves would have been more efficient, less set-up, and more robust because now we not only require internet connectivity, but must also navigate the Twitter API OAuth functionality, respond to any changes in the Twitter API, and have behaviour in place to recover from any Twitter errors from simple over capacity to more subtle error parsing..

Resilience to Error

Does your application code look like this fragment of C#?

```
static void Main(string[] args)
{
    try
    {
        doStuff();
    }
    catch (Exception)
    {
        shutdownAndRestartApplication();
    }
}
```

This feels like (and is) a hack. For non-programmers let me explain. The main behaviour of the application takes place

within the 'doStuff' function and following the guidelines discussed under Robustness heading the code therein would take careful account of potential error scenarios and deal with them appropriately. But what happens if despite one's best intentions an unforeseen error occurs? This may be an error never encountered in testing or an error there was insufficient time or resource to address. In that case the 'shutdownAndRestartApplication' function kills the current application after restarting a replacement. Careful use of state logging and recovery may mean that the application can restart in approximately the same state as before the unexpected error fired.

The Family Archive, a previous demo developed by Shahram Izadi in our group made extensive use of state logging which, combined with a fast start-up time meant that the application could crash and restart without anyone noticing!

Photobox does not achieve this, during start-up it needs to establish connectivity to the Bluetooth printer, the authentication on the Flickr API, and the authentication on the Twitter API which all take time, but it does restart when an unexpected error occurs and this has proved useful in the field.

Testing

Testing slow technologies raises another concern. There are established methods for testing the logical and other features of code including load and stress testing. There are also ways to test how the software will behave at future dates, for example those employed during the Y2K recoding exercise. But without running a system for year after year it is difficult to know for sure how the interplay of software, hardware, and other factors will affect the performance of the system.

Power Management

Slow Technology need not be always on technology. Indeed in the case of Photobox interaction, either status reporting or printing is spasmodic. Our original design had the Photobox application implemented as a service which would be run on start-up and then put the computer into standby when requests to print or to report status had been checked. The computer OS' scheduling system could then be enlisted to wake the machine every hour, day, or any chosen period. Such choices, and other more sophisticated power management features seem ideal for slow technologies.

CONCLUSIONS

Many software engineers would read this workshop submission with some incredulity. They would point out that all the issues I raise are in fact solved in other branches of software engineering. In our everyday lives we are surrounded by software which has been engineered to robust and resilient standards and that can stay functional without error over large time periods. These range from the mundane (the software in your TV set-top box) through to the large scale (the control software for a power plant). Indeed recent advances in cloud provision of computing infrastructure have led to a more public exposure of attainable uptime and performance target.

But in the 'HCI Lab' and within HCI research deployments such levels of engineering are difficult if not impossible to attain. We are used to building demo quality applications, or even applications that may be deployed in a limited trial, but the jump to production quality and robustness is significant and I would welcome discussion on how we design the infrastructure for slow technology and how we can pool engineering experiences to make such robustness goals more easily achieved.

REFERENCES

1. Odom, W. Selby, M., Sellen, A., Kirk, D., Banks, R., Regan, T. 2012. Photobox: on the design of a slow technology. *DIS '12*.