Symbian - a post mortem

Mika Raento, mikie@iki.fi, 2012-10-13

(You could actually call this a S60 post-mortem since I mostly write about what Nokia did with symbian, but UIQ never amounted to much outside SE's dreams).

This is an extended good-bye piece for Mika Raento's blog on Symbian. Since this is a post-mortem, it focuses on what went wrong rather than on what went right (first smartphones, extremely good power management, first cameras, some nice hardware). This has also been a long time in writing (since 2010), not very topical today (late 2012) but I think still an interesting historical perspective.

History

Caveat: I wasn't there. I might be well off some of the points, but I think the approximate train of thought at Nokia I portray is correct.

Nokia chose Symbian in late 1990's - an eternity ago. They were targeting something like the 9210 Communicator and the 7650 - devices with 8 (4) megabytes of RAM and 4 megabytes of disk (the 9210 did come with an additional 16 megs of disk in the form of an MMC card). At that point in time there weren't that many options. Linux wasn't really viable for such specifications. Windows CE was around - but let's face it, it's still not a great choice, let alone then. The previous Nokia smartphone (not that the term was widely used yet) was the 9000/9110 Communicator which ran GEOS (of all things), and was probably seen by Nokia as a dead end by then.

What people at Nokia saw in the late 1990s was a Psion 5. Psion 5 was an instantly-on, multitasking, lightning-fast clamshell PDA running its own 32-bit unicode-ready operating system. It had a suite of productivity apps and a devout following. It probably looked very, very good at that point. So Nokia went with the Psion EPOC OS, later named to Symbian. Psion 5 ran EPOC release 5 (R5), the Communicator would run 6.0 and the 7650 6.1.

The technology

Poor ~~third-party~~ application performance

Now let's go back to the Psion 5. Its performance derived from three major design points: execute-in-place ROM, battery-backed RAM as storage and monochrome/grayscale screen. What do these mean?

Execute-in-place ROM means that there is no 'loading' of programs. No reading from disk. No paging needed. Size of the executable doesn't affect performance (though it would affect the amount of ROM needed). No actual dynamic linking as although there were DLLs (for sharing code between processes) everything the linker normally does would have been done at ROM build time. No relocations. Starting an application meant creating the kernel

structures for a process and jumping into the right memory location. The dynamic linker's performance didn't really matter. Even virtual memory was only sort-of as a number of processes were loaded into fixed physical memory locations so that switching between them didn't incur a TLB flush, so switching between the built-in processes was pretty fast (and there's a lot of switching since Symbian is a micro-kernel operating system, meaning most operating system calls would incur a process switch).

Battery-backed RAM for storage meant that storage was blindingly fast (well, for those times, the memory bandwidth and latency would not be anything to write home about today). No disk seeks, no fetching across buses. For example the Epoc (Symbian) native database access would have been instantaneous for all intents and purposes. Probably the file server was designed around RAM speeds. Storage access patterns, amount of data and number of writes didn't really matter.

(A smallish) Grayscale screen meant that drawing things on-screen didn't involve pushing around too many bits. In addition, graphics assets were in blitting-friendly format in ROM so 'loading' a bitmap just meant giving the right memory address to the font-and-bitmap-server. The number of bitmaps you used didn't cost anything (performance-wise - of course it would again cost ROM size).

Now consider instead a third-party application on a 7650. It's stored in the internal flash. Starting the application requires loading all of the executable code from flash into RAM (demand paging didn't come until late sw versions of the N95). After that it would need dynamic linking fixups. Bitmaps were now color, and needed loading from disk into memory first. Writing to disk through the Symbian database was slow as molasses.

Things were exacerbated by the fact that all the example code and documentation was written without taking into account the completely changed performance characteristics of the platform. You were encouraged to load all your bitmaps on startup. The platform code for loading a bitmap would open the file, read the bitmap directory and seek to the bitmap in question for each bitmap you loaded, multiplying disk accesses (we had to write our own bitmap loader for Jaiku to get acceptable performance).

And things got worse over time. Nokia gave up on execute-in-place ROM, probably for cost reasons. Resolution and color depth increased, meaning slower loading (and blitting) of bitmaps. Bitmaps were replaced with SVG (scalable vector graphics), which meant that showing a bitmap wasn't just blitting from a memory location, it meant loading from disk, parsing XML, building an internal representation of the graphic and rendering that into a hi-color bitmap. Lazy-loading by the platform would have been pretty neat.

On early MMC-supporting devices it was pretty easy to lose data. We corrupted a third of a 100 cards in a couple of weeks writing log files to the card when running the Reality Mining experiment. Obviously FAT on a removable flash card on a battery-powered device was a recipe for disaster. To 'fix' this Nokia removed most write caching in 3rd ed. Writing to the MMC was now 50 times slower (for small writes) than to the internal memory.

Obviously you could write performant apps for Symbian - there are very good games out there and Google Maps for Mobile is (was?, it's no longer distributed) pretty cool,

performance-wise (although I worked on it the basic architecture was in place before I joined the team and has nothing to do with me). But these got the performance by basically discarding all of the platform components and replacing them with their own (graphics loading, UI components, data storage). Not the immediately obvious answer when approaching a new platform. And the performance of Nokia's own apps still sucks on my E72.

Shit Java startup performance

S60 phones weren't actually a horrible J2ME platform (if there are such things as non-horrible J2ME platforms). If you don't think so, try Opera Mini - it's nothing short of amazing. But starting a J2ME app takes ages - long enough that it was a deal-breaker for many people.

Non-standard OS, compilers and libraries

Don't get me wrong. People are willing to learn new things - witness having to use Objective-C on the iPhone. But not being able to use almost any of your existing libraries or skills is different.

Symbian did not come with a reasonable C library. There was one that had been used to port the JVM but it only contained what the JVM needed. This was alleviated later, much later - more on that further down.

You were not meant to have globals in Symbian programs. Before Symbian OS 9 it was almost impossible: applications were not EXEs but DLLs loaded into a platform process, and the toolchain tried to make it impossible to have global data in DLLs - in the name of not having to have a page of writable memory for each process loading the DLL (remember that you could run a process with 4k of stack and 8k of heap - an extra 4k for a DLL was quite a lot).

Symbian APIs were asynchronous, as a rule and you were not meant to use extra threads (again, to save the 4k of stack you'd need for a thread). Some async APIs you could explicitly wait on to mimic synchronous calls but some you couldn't. Since documentation assumed you weren't using threads, it didn't tell you which APIs were thread-safe. Asynchronous APIs meant that all of your nice linear logic needed to be turned into state machines instead (some people like this, witness node.js, twisted - but you can write much simpler async code in languages with dynamic typing and lexical closures).

For a long time the standard Symbian device compiler was GCC 2.9 (from 1998!). You could use ARM's compiler instead, but that cost several thousand dollars per seat. Emulator builds were done using either Metrowerks CodeWarrior which was also stuck in the 90s when it comes to C++. Visual Studio compilers were a possibility, but support was dropped later.

Until Symbian OS 9 C++ exceptions were not supported. This had been decided by Psion in the mid 90's as they thought (probably correctly at the time) that exception support was too flaky and too expensive (in code size and run time). So you couldn't use C++ code that used exceptions or stack unwinding for correctness (like the standard library).
So no decent C library, no globals, no threading, no exceptions, ancient compiler. The

chances of getting some useful third-party library or your own non-Symbian C or C++ code to work out-of-the-box were pretty slim and often it was infeasible to make the necessary changes.

C++

There is something fundamentally broken about building reusable components out of C++. Most attempts are abject failures - some spectacularly so (such as Taligent). There are some exceptions (boost, maybe Qt, maybe MFC) but even they tend to be expert-level only (compared to re-use in python or perl, which everybody does), and definitely backed by more resources and experience than was applied to Symbian.

Testability

It was pretty hard to make Symbian programs, especially user-facing programs, testable.

There was no real unit-testing framework shipped with the platform (which is not that unusual, but it was much harder to get existing frameworks to work well with Symbian). The platform APIs were a shitload of concrete C++ classes - you couldn't mock them without wrapping them. The Symbian UI was framework-oriented, rather than library-oriented: your code was often called by platform code rather than you calling platform code, so to test your code you needed to mimic the platform call sequences which were brittle and asynchronous (call this method, then let events run for an unspecified time, then call this method). Application startup was done in closed-source platform code so getting UI code to run inside a test framework was black magic. Writing testable code meant doing a *lot* of up-front work. Build times were pretty bad. Using Symbian's own toolchain it was pretty much impossible to get builds under half-a-minute. The build ran a BAT file that ran a perl script that generated recursive makefiles and then ran make. The emulator took longer and longer to start up as the platform grew (the 5th edition emulator took over a minute). It was non-trivial to load new code into a running emulator (it was possible in theory, but for example Nokia's own IDE keeps DLLs loaded into the debugger so they can't be replaced). IDEs helped but Nokia eventually phased out all the good IDEs (see 'IDEs' below). That sub-second compile-run unittests cycle you so like? Forget about it. I was eventually able to use scons-for-symbian + my own semi-headless test runner in just under 10 seconds on a 4-core 12 GB RAM machine - a machine you wouldn't have been able to get in mid 2000s when Symbian was still dominant.

Lots of interesting APIs only worked on the device, worked differently on the device than on the emulator, had very different performance on-device or worked differently on different devices. These included internet connectivity, bluetooth (you needed to get a specific no-longer-manufactured USB dongle to get it to work at all), low-level memory management (heap and stack sizes, memory protection), dynamic linking (emulator uses the windows native linker in the end), camera (some emulators emulated the camera, some didn't, the emulated camera app was different than the one on the phone), platform security (the emulator didn't care about many things the device would), application installation (there even was a web site called "Why the fuck won't my SIS file install?" that debugged your SIS file for you), operating system software release (all devices forked the platform produced by the platform team at Nokia and the SDK was another fork), removable media, built-in media,

voice and data calls (which never worked in public SDKs). You needed to test your application extensively on multiple devices (there would at any point in time be up to 30 devices on the market). Automatic testing on-device was difficult (for example fully-automatic installation was frowned-upon because of platform security) - we didn't consider it worth the effort at Google (though Symbian was never highest-priority at Google due to the minuscule amounts of traffic it generated).

IDEs

In the olden days you could write your Symbian code in Visual Studio. Now say what you will of Microsoft, Visual Studio is pretty damn good when it comes to performance and stability. Especially Visual Studio 6 was very, very performant on the machines that we had in early-to-mid 2000s. You pressed F5 and the emulator started up pretty much instantaneously with your newly-compiled code. You could also use CodeWarrior, but that cost more and was pretty bad compared to VS. Since Microsoft and Metrowerks compilers' LIB files and C runtime aren't compatible, SDKs came in two flavours: VS and CodeWarrior.

IDE support back then was implemented by a toolchain target that created an IDE project from your Symbian-specific MMP project file. This obviously meant that if you needed to change the project (add a library or a source file) you took a bigger hit as you needed to run the Symbian toolchain and reload the project. The toolchain also didn't grok dependencies between your components, so those had to be added manually (for VS you could use ready-made dependency-management plugins and for CodeWarrior we rolled our own). If you had client-server code then you needed to hack something together to get the server to build as needed.

At some point Nokia decided that they really couldn't live with a dependency to Visual Studio (I heard that they didn't want such a strategic dependency on Microsoft. Hah.). So they dropped Visual Studio support and bought some rights to CodeWarrior so they could use it in perpetuity (and sell it - later give it away). This made things quite a bit worse. CodeWarrior's performance was not as good as Visual Studio's, its debugger was worse and in general the IDE left a lot to be desired (e.g., it doesn't treat files as paths - instead it has a set of 'search directories' and a set of 'filenames' and it looks for the files in the search directories - guess whether you can have two files with the same name but in your source tree?). And the CodeWarrior project file creation was broken in several ways.

A side-note: now that Nokia owned the Symbian version of MetroWerks compilers they were also the only ones maintaining them. Which they really couldn't do - seems they had neither the will nor the skills to do this. There were almost no new versions of the compiler and the changes were absolutely minimal. More on software skills later.

Even Nokia realized that CodeWarrior was not really the future so they decided to hack on Eclipse to make it work with Symbian, calling their fork Carbide. Open-source, you know - the new hotness and strategically safe. Again, however, they lacked the skills to do that well. The first version tried to do the right thing and have the IDE do the builds, but I assume they were never able to get it to work reliably, so later they just let it run the command line toolchain. So build times, rather than being improved by the IDE, got worse. We at Jaiku actually sat down and talked to the heads of the Carbide project at one point after getting

very frustrated with it (we ended up using CodeWarrior, that's how bad Carbide was). Of the two heads we met, neither of those guys was an actual coder. How do you run an IDE development project if you don't know what an IDE is supposed to do?

Graphics

Although graphics could be seen as part of the Technology, its impact was big enough that it warrants a chapter of its own.

When the 7650 came out, its graphics were a reasonable match for market expectations. It used colors sparingly, had a very small number of UI concepts (lists, grids, soft keys, menus, selection boxes, editors and a couple of others) and was a step up from the dumbphones/featurephones of its day. But sadly, it also pretty much reflected the *maximum* capabilities of the UI technology that Nokia could create. Later we would get themes (background images and colors that the user could choose), and much, much later fancy transitions.

Nokia didn't have the experience or expertise to improve on the UI APIs. Whereas the iPhone builds on the world's longest continuing tradition on the best APIs for UIs (Macintosh, NeXT, OS X, iOS), Nokia had a bunch of UK engineers for the low-level stuff and a bunch of Finnish engineers for the S60 UI layer. They couldn't write UI libraries and APIs that would allow internal or external developers to step up their game. The worst example of this is looking at the code for each UI element in the 5th edition source: say a button's click code has three times as much code doing the animation as there is for handling the actual click. Apple's API separates animation from other UI code and especially simpler animations (moves, fades) are declarative (update 2012-10-26: link fixed).

In 2007 Nokia brought out the N95, which had 3d acceleration onboard. But they didn't add 3d acceleration to all their phones, just a few. This meant that, unlike Apple, they couldn't build their UI capabilities and performance on hardware acceleration.

Browsing

The first S60 phones shipped only with a WAP browser (remember WAP? operators thinking you'd pay-per-click for their walled-garden web?). It was definitely not possible to run a full 2002-web -capable browser in 4MB of RAM, though there were some limited third-party browsers (like Doris).

In (roughly) 2006 Nokia started shipping a WebKit -based browser on the S60 3rd edition phones. They had managed to build an ambitious browser team including (AFAIK) some original KHTML/WebKit engineers. This was a great engineering effort and resulted in a modernish browser (though with many rough edges, you could call it a beta).

Then they lost the team (exhibit 1: http://www.linkedin.com/in/dacarson , exhibit 2: http://www.linkedin.com/pub/antti-koivisto/11/904/5a6) and shipped the same beta browser for two years (see symbiatch's short article and the history of browser versions).

In 2007 the iPhone jumped way ahead with a browser that could be built with better graphics

libraries and HW, much more memory, a much larger screen and a good touch screen. S60 never caught up. The iPhone 1, although shipped in much smaller numbers generated much, much more traffic than all the S60 models combined.

Process and politics

(This is somewhat more speculative as I never worked inside Nokia or Apple, but is based on pretty educated guesses).

Nokia and Symbian

Until Nokia finally bought up Symbian, the OS development would go something like this: Nokia needs feature X (say Bluetooth or Wi-Fi). Symbian says OK: it's gonna be in release 'next + 2', where 'next' is what Nokia wants to ship X with. So Nokia builds X themselves and for a few releases they go with their own stuff, then they make the painful transition to the Symbian one. In addition of paying for stuff from Symbian they need to build it themselves + pay Symbian + do a transition. Also: Nokia's OS skills are not a good as Symbian's so their version is pretty bad. Also: the transition sometimes breaks third-party apps.

Symbian also made their own UI, called Techview - which was meant to be just a placeholder that licensee's would replace. Ericsson had UIQ, which Symbian was more closely associated with. Nokia built S60 themselves (and S80 for communicators and S90 for their failed TV-enabled device). All of these had a similar set of underlying APIs, but also significant differences. Nokia needed to build a separate set of applications for each of its own platforms and third-party developers needed to support each UI platform separately, fragmenting their efforts.

Product management and engineering at Nokia

AFAICT Nokia's engineering was very much top-down. Design prototypes were turned into phones, applications and features. These were broken down to work to be done by teams, and the teams had very little say in whether the stuff they ended up doing made sense.

An example: S60 had a built-in e-mail application that had a feature for checking for new e-mails automatically. It would create an internet connection via GPRS/3G when first requested, and then poll for emails periodically. All implemented nicely by some engineers. However, if the connection was ever dropped, the polling just stopped. Which meant that in reality there was no automatic checking for new e-mails, since you couldn't rely on it. I find it highly unlikely that the engineers who built this wouldn't know this, but clearly they had no way of influencing the end result. The same problem remained for several iterations of the platform.

This is in stark contrast to the visible effects of the engineering culture at Apple. There clearly the emphasis is on getting the end result right, whatever teams and layers need to be involved. The way the browser, Wi-Fi and internet connectivity for example know of each other to tell the user why they can't connect is miles better than S60 (or most Microsoft products).

Nokia seemed to care very little about issues with existing phones - only the next phone mattered. They did release new software versions for phones to deal with the worst bugs that operators complained about, and from the N95 on there would be significant improvements to existing phones (culminating with Anna phones being upgradable to Belle).

Openness and developer relationships

A 'normal' third-party developer got an SDK from Nokia. This came with so called 'public' APIs. Both Symbian and Nokia also had 'Partner' APIs, which were (of course) physically present on the phones but had no headers in the SDKs (and later no LIBs either). These non-public APIs were needed for lots of interesting things, like getting the current cell information for locating the phone. Things would later improve some, but the distinction remained for a long time. It was possible to become a partner. With Symbian it was mostly a question of money, but with Nokia you needed to prove your worth to their partnering organization. If you talked about the non-public APIs in their forums, they would remove the posts. (To be fair, Apple does the same thing *and* required an NDA just get the SDK).

Nokia didn't have a formal bug reporting program (to start with) and no public listings of known issues (this, BTW, is pretty much the same with Apple, but in stark contrast to Microsoft). Later they did announce a program, but at least I never got a resolution to any of the bugs I reported. Bugs reported against Carbide or the Metrowerks compiler would sit untouched, then closed when the next version came out even though the issue was not fixed. (I cannot link to the bugs as the bugzilla server no longer exists).

Symbian used to ship source for some parts of Epoc, Nokia stopped even that. The source was often very useful (the same way opendarwin source is sometimes useful :-).

The S60 Platform was originally tested with only Nokia apps, and shipped when only just working with those. Messages would not appear in Inbox if accessed by another app at the wrong time, bluetooth stack was flaky in repeated use, network stack was flaky with repeated use, using too much memory would freeze or reboot the phone. This probably got at least somewhat better over time, as they did start testing also with 3rd party applications.

Conclusions

The S60 platform clearly tells the story of a great hardware company struggling to become a software company. I think it also tells a story of how hard it is to build expertise in software - that without a crucial mass of people, companies, products and projects in an area (in this case, specifically UI libraries and compilers) you just can't make. The Bay Area has that mass - Finland doesn't.

There are shades of the Innovator's dilemma here: When you have almost 100% market share how do you know you need to completely recreate your technology? The iPhone started from 128MB of memory, 16GB disk and accelerated graphics - a system built for 4MB/16MB and bitmap graphics could not be scaled to the same user experience (or developer experience). (Jeff Dean has often said that you can design a system for 10x growth, but 100x requires a different system)

Nokia most likely had a cultural/organizational/managerial problem dealing with both of the problems above. They did try though: Meego (and they had a couple of linux-based prototypes even before that), OpenC, Carbide, Qt were all attempts to both make significant improvements in the current offering or to create the next technology. It ended up being too little, too late.