medium.com /@benbob/a-tale-of-two-operating-systems-8f3ffd4f972e

A Tale of Two Operating Systems

Ben Fathi

14-18 minutes

"Reach" vs. "Rich": Strategies for platform growth and ecosystem development

> "It was the best of times, it was the worst of times." — Obligatory Dickens quote. A Tale of Two Cities.

A few months ago, on a whim, I re-published a blog I'd written last year, entitled "What Really Happened with Vista: An Insider's Retrospective." To my surprise, it received well over 100,000 views within a couple of days. I had to go back and see what all the hoopla was about.

The content hadn't changed. Social media networks had just picked up on it the second time around. I realized upon a second reading that while I'd spent a lot of time explaining the problems with Vista, other than citing the inherent benefits of open source, I hadn't really offered too many solutions to the challenges I'd listed.

So here's an attempt at a better follow-up and conclusion. It's not about Vista but rather about operating systems in general: what it takes to build a successful platform and the ecosystem around it and how the health of that ecosystem ultimately dictates the success or failure of the platform in the long run.

An operating system shouldn't be viewed as a standalone piece of software that a company ships once in a while. It is, instead, the sum total of the entire ecosystem of solutions that comes to life around it. Over the long haul, the platform stands or falls due to the health and vitality of that ecosystem.

A successful platform lasts decades. As such, every architectural decision made early in its life cycle becomes critical in the long run, much more so than any dozen decisions made later in its life.

When analyzing the recent history of operating systems, a pattern emerges that is worth scrutiny. At any given point in the recent past, in fact, *two* operating systems have competed for dominance in the general-purpose computing arena. Examples are: Unix vs. Windows, Netware vs. Windows, MacOS vs. Windows, Linux vs. Windows, iOS vs. Android. By dominance, I am referring to some fuzzy formula based on unit volume, total revenue, application availability, total cost of ownership (TCO), performance, reliability, and developer mind share.

The names have changed over the years but the story is often the same. When it comes to building a platform, there are basically two strategies available: *rich* or *reach*. (I didn't invent these terms, they've been around for decades.) Both can be successful but they outline markedly different approaches to ecosystem development.

In the case of the former, a ***rich***, coherent, and elegant end user experience is prized above all else. Apple's products fall squarely into this category, where a "walled garden" approach delivers a superior end-user experience at a premium price, achieved by tight control over the ecosystem, a reduced number of choices at each integration point resulting in a smaller test matrix and hence fewer opportunities for breakage. This approach can be highly lucrative; the iPhone covers less than 15% of the smartphone market in terms of unit share but accounts for 90% of the overall profits.

By intentionally reducing the available hardware choices and maintaining complete control over the out-of-box experience, Steve Jobs forced the Apple ecosystem to "move up the stack" and innovate in

applications instead, while also improving overall product quality and customer satisfaction. Combined with a central app store, this also resulted in better quality control at the application level.

In the case of the latter strategy, the top priority is to ***reach*** as many customers as possible. Windows falls into this category, as does Android. Microsoft's mantra of "a computer on every desk and in every home" was the rallying cry for Windows for many years. By necessity, the *reach* strategy requires an open ecosystem of partners and many integration points into the platform. It often yields a less cohesive end-to-end user experience as various integral parts of the product are designed and tested by different vendors.

Windows-powered laptops are available from dozens of vendors in a dizzying array of hardware and software configurations. When your platform strategy is to *reach* the broadest audience, these OEMs (Original Equipment Manufacturers) often end up competing on price. They choose the cheapest available piece of hardware to save a few pennies on COGs. In fact, a given model from any manufacturer may incorporate different hardware and software components on two different days of the same week.

The OEMs have to make money somewhere in the value chain, so they also add what has euphemistically been called "crapware" to the system: everything from antivirus software to photo editing apps and more. As a result, no two Windows-based systems are identical.

Your PC shows up with a few (or a few dozen) third party apps pre-installed, apps by vendors who paid the OEM a fee for that placement — hoping to sell you a monthly subscription for features ranging from antivirus to backup to ink cartridges. Given the historical lack of application life cycle management in Windows (more on that later), these apps often step on each other's toes. As expected, the end to end quality suffers.

With a MacBook or iPad, available only from Apple, you get the choice of *one* disk controller, *one* network controller, *one* graphics controller, etc. — whichever one Apple decides to offer. It wouldn't be too much of a stretch to claim that, often, the only thing you get to pick is the color of the device: silver or gray. Apple intentionally reduces the hardware interoperability matrix which, in turn, improves quality. In addition, no third party apps are pre-installed on the system and quality control is enforced through the App Store.

There is no magic here: increased test combinatorics and distributed responsibility for end-to-end testing *always* result in lower end product quality. Windows (and Android) will *always* deliver a less polished product than anything Apple produces since someone else (the OEM) gets to pick and choose the hardware and software components that make up the overall system — and he gets to do so from among a dizzying array of options, most of which haven't actually been tested together. This is exactly why Microsoft started designing its own "Surface" tablets and Google its own "Pixel" smartphones.

The Windows NT kernel natively implemented multiple user identities as well as proper security isolation boundaries from day one. Dave Cutler and his crew had built an elegant modern kernel capable of running applications in one of several personalities: Win32, POSIX, and OS/2. The resulting kernel has fueled all operating system work at Microsoft since its inception 25 years ago.

Over time, the Win32 subsystem became the de facto personality as it enabled compatibility with Windows 95 applications. Unfortunately, this also brought with it a single-user "personal" computer operational model that bestowed administrative privileges upon the user, allowing her to read or write all files and registry entries by default. Proper controls were available to lock down a machine and limit administrative access (and these were used on Windows Server editions) but they were not utilized on consumer machines. This was the equivalent of installing the most secure locks on your home (in the kernel) but then leaving the front door open.

Note that this was an architectural decision made in the early nineties, long before the internet existed. What may have been a sensible simplifying decision when the first PCs shipped was no longer reasonable for a new generation of computers connected to the internet.

Additionally, the Win9x operating system didn't have an application life cycle model built into it. Installing an app involved copying some files around, possibly installing a device driver in the appropriate system folder, and writing some values to the registry.

This model was abused by applications and device drivers (both third party and Microsoft's own) for many years, resulting in massive quality issues. If you installed App A before App B, you would get one behavior. If you installed in the reverse order, you got a different result. Apps and drivers stepped on

each other's toes, often unknowingly, writing over shared files and registry keys and even kernel memory, resulting in untested scenarios. Uninstalling apps was also fragile and similarly left the system in an unknown state.

That problem alone meant you could never be sure about test results. To successfully test any given system configuration, you had to install Apps A and B *both ways* and run all the tests twice! Three apps, seven variations!

Multiply that by thousands of apps and drivers across dozens of Windows Service Pack releases and you understand one of the root causes of the quality problems in the Windows ecosystem. Microsoft had certification processes to help catch many of these problems but most consumer apps didn't go through the process — and the process didn't necessarily catch problems caused by "cross-application" interference. Regardless, the end result was a fragile ecosystem.

The inherited Win9x "single user personal computer operational model" also made it easier for viruses to wreak havoc on the system as any piece of malware downloaded from the web could easily gain control over the entire machine due to the lax security defaults.

When we tried to improve security in Vista, we *necessarily* broke application compatibility. Most often, the "compatibility" we broke was the set of assumptions these applications had innocently made about what they could and couldn't do in their installation scripts and running code. *If I can read and write everything, there are a lot of shortcuts I can take.* The Win9x operational model, by being overly permissive, had made it so easy for apps to misbehave that they almost inevitably *did*.

The same broad ecosystem that helps a *reach* platform grow rapidly also slows it down in the long run *if and when* you take shortcuts. We had missed the opportunity to enforce proper security isolation and application life cycle management in Windows in the early days — and both decisions would come back to haunt us and our partners for many years to come.

The lesson here is that you have to be ultra careful with architectural decisions early in the life cycle of a platform. They carry much more weight than any decision later in its lifetime.

> "Those who cannot remember the past are condemned to repeat it." — Agustín Nicolás Ruiz de Santayana y Borrás. 1863–1952.

More than five years elapsed between the release of Windows XP and that of Vista—including the Longhorn detour. Meanwhile, the world outside Redmond had changed drastically. The internet had taken off like a tornado, upending everything in its path. Simultaneously, the interesting form factor had evolved; the smartphone had arrived on the scene.

Microsoft was still spending most of its energy fighting the last war — the desktop and laptop generation war. That generation of computing and its demands had consumed us as an organization. We would spend an additional three years building Windows 7, a massively more popular release with much better end to end quality and application compatibility, but one that was still targeted at the old world instead of the new one.

At that time, the Windows Phone team was not part of the Windows organization. Instead, it was owned by a tiny team in another division and was based on the 20-year old-Windows CE (Consumer Edition) code base which predated even Windows 95. Multiple attempts, including several acquisitions, were made later to try and jump start Microsoft's presence in the smartphone space but none of them paid off in the end.

The lesson here is that applying the constraints of one era of computing to the next is a mistake best avoided. Simply put, there are too many architectural changes from one generation to the next. Given the rate of change in this industry, that would be like trying to fight the US army with bows and arrows. Using Windows CE to fight the smartphone war and using the Win9x operational model in the internet age are just two examples of such a mistake.

Carrying the tools of the previous generation into a new war will eventually backfire, adding so much support and compatibility burden to the shared code base that it becomes a ball and chain around your ankles. That, in turn, becomes not only a resource drain on the organization but also comes to define its character, its very structure, and — ultimately — its priorities. Innovator's dilemma in a nutshell.

There were so many simplifying assumptions that we could not incorporate into the Windows architecture because it had to continue to support all the old hardware and all the old apps, apps that

often didn't even make sense in the new world. Our very success as a platform was the root cause of our inertia as a team.

Application compatibility can be a great boon to a new operating system as it tries to gain traction but it should only be offered in a way that does not compromise the architectural integrity of the platform. It is often better to start with a clean sheet of paper when designing for a new generation, applying the lessons learned from the previous generation, but targeting them at the next generation.

Windows does not play in the smartphone operating system market today. The current battle is between iOS (rich) and Android (reach). As Mark Twain is supposed to have said, history doesn't repeat itself but it sure does rhyme.

Here, finally, is my "Top Ten" list of Do's and Don'ts for platform development teams. Not surprisingly, some are equally applicable to *all* software teams:

- **Do**: Pick your ecosystem strategy up front and stick to it. Chances are you won't be able to switch down the road if you start with *reach*.
- **Do**: Protect the architectural integrity of the platform and the health of the ecosystem like your life depends on it. In a sense, it does.
- **Do**: Constrain choice artificially at lower levels of the stack in order to push the ecosystem up the chain. Choice is the mother of complexity. Every time you offer a choice, you double the test matrix.
- **Do**: Plan your releases top down and not bottom up. The former results in a more coherent whole while the latter will result only in incremental improvements offered up by each sub-team protecting its turf. **Don't**: Ask each sub-team to "come forth with your plans as a starting point".
- **Don't**: Allow any feature to hold a release hostage. The train leaves on time. If you're not ready, that's your problem — not everyone else's. **Corollary**: Ship often.
- **Do**: Share code where it makes strategic and architectural sense, not where it's organizationally convenient.
- **Do**: Break compatibility if and when it makes strategic sense. **Don't**: Let compatibility alone be "the trump card" in any discussion; it's only one input into the planning process.
- **Don't**: Ship any software on a device that you can deliver as a service from the cloud. The latter has proven itself to be a much more sustainable model of software delivery.
- **Do**: Add instrumentation and telemetry to every facet of the platform. In this day and age, you don't have any excuse not to. Besides, you can only improve what you measure.

I'm sure I'm missing quite a few and would be glad to amend the list. What do you think?