Events in Flux: Software Architecture and Rhetorical Subtraction

Andrew Pilsch

My goal today is two-fold: mainly, I'm going to talk about Facebook's software architecture as a model for programmatic *detractio*; however, I am interested in this question, which I'll talk about later, because I think what Kevin Brock has called "rhetorical code studies" has an important, maybe even paramount, place in the tools of a digital rhetoric. To go back to the ancients for a second, as we all probably know, education in grammar was the foundation upon which a facility for rhetoric was built. In this example, we find a primary or more foundational (if we can use a hierarchical spatial metaphor I'm deeply suspicious of) discourse of bare language as being a precursor to the more complex, nuanced, and gymnastic work of rhetoric. I think we can suggest that grammar, for the ancients,

was training in a facility with language and rhetoric was the facility in artfully deviating from or manipulating those conventions to produce eloquence.

I want to ask after Facebook's Flux architecture today because I want to start mapping the equivalent to grammar for a digital rhetoric. If we, as Doug Eyman argues, can map from classical rhetorical figures and tropes into the digital domain, this implies, in my grammar-into-rhetoric history of rhetorical training, that there is some kind of artful deviation happening in the production of digital, argumentative eloquence. So, if we want to follow out this analog-to-digital map I'm making, there would imply that there is some kind of baseline or conventional digital ground that is being refigured by the tropes and figures of digital rhetoric. While you could probably be correct in just saying "Andrew it's still just language," and we probably wouldn't be wrong (and I could just sit down), I want to ask if

things like the base materiality of computation: protocols, languages, and, today, architectures don't also come to constitute the baseline equivalent to a digital grammar from which a digital rhetoric then deviates in order to produce the desire-called-persuasion.

So, if I'm right, things like TCP/IP, the x86 computational architecture, or the JavaScript programming language (just to pick a few of my favorites) might play important parts in framing digital rhetoric *and* that we as digital rhetoricians should be paying attention to the communicative effects inherent in these really dull, often obscure technical systems. I think such an approach is especially relevant, as I discuss today, to the emerging conversation around "platform rhetorics."

A platform is a software architecture (or probably more accurately a software ecology) that enables the building of other services. Facebook, Amazon's AWS Cloud Computing,

Google's wide array of services, and Twitter are all examples of platforms. In *Platform Capitalism*, Nick Srnicek defines platforms as "digital infrastructures that enable two or more groups to interact" and through their leveraging of network effects constitute "an extractive apparatus for data," akin to economic entities in capitalism like futures markets that do not produce commodities but instead extract profit from existing flows (Srnicek 43–48).

Rhetoric happens on platforms, no doubt; these interactions are often discursive (though many platforms such as those used by Monsanto and GE have little to do with social media) but they are also, importantly for Srnicek, a key investment in the formation of economic and political power, as we know all to well in 2018. **Slide** While I talk through my approach to platform rhetorics, I want to leave this quote from Doug Eyman

up, to ask us to think about what is *digital* in the digital rhetoric of platforms and what is analog:

> While rhetoric provides the primary theory and methods for the field of digital rhetoric, the objects of study must be digital (electronic) compositions rather than speeches or print texts. This is not to say that scholars of digital rhetoric may not make connections between analog and digital objects or focus on the cultural and socio-historical circumstances that lead to, influence, or are imbricated with the construction of digital texts, but that the primary boundary condition for the field is the distinction between analog and digital forms of communication. (n.p.)

In the introduction to their recent special issue on platform rhetorics, Dustin Edwards and Bridget Gelms declare the focus of the issue on "new rhetorical contexts we currently face on platforms" and the contributions generally focus on what rhetors do on platforms and how platforms are used by these rhetors to persuade. And that's all well and good, but aren't we, at least a bit, putting the cart before the horse when we do that? "Behold, rhetoric," we say, pulling a dust cover off a computer. But as

Nietzsche, who I hope y'all get I'm referencing here reminds us, "If someone hides something behind a bush, looks for it in the same place and then finds it there, his seeking and finding is nothing much to boast about" (Nietzsche, pg. 147).

I don't want to sound like I'm knocking the work done so far on platform rhetorics. Instead, what I'm trying to suggest is that if our analysis of platform rhetorics starts from the assumption of rhetorical users and precedes toward the platform, I worry that we miss most of the story about what makes these things novel. Humans have rhetoriced for a while now, but platforms are, as Edwards and Gelms make clear, totally novel concept. Shouldn't we start with the novel parts and, instead, bracket the humans for a while? (Btw, I'll be selling "shouldn't we bracket the humans for a while?" T-shirts after this panel).

Toward that end, I'm going to use the rest of my time to discuss the rhetoricity of Facebook by starting from its code

architecture, which is called Flux, and building on that. Maybe by the end we'll get to some actual humans sharing actual doggo videos, but I make no promises.

So, as you may or may not be aware, software is formed, shaped, and structured by an analogy to architecture: thus, like architecture, software can be said to be composed of certain functional patterns and part of designing software (to say nothing of making it) is organizing the components of a larger project according to patterns that work (and avoiding the dreaded "anti-pattern"). **Slide** The early work on this was done in the classic software engineering text, *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, John Vlissides, Ralph Johnson, and Richard Helm, in case you're interested. This work was incredibly important for managing how the era of object-oriented programming was inaugurated (think languages like Java or even Ruby for

examples of OOP). This is because in OOP, software is composed of a series of objects that interact in limited ways but mostly keep to themselves. These interactions can be structured in a variety of ways; *Design Patterns* sought to formalize the good ones.

Slide In front-end web programming (which used JavaScript's limited OOP support quite heavily in the early 2000s), one of the major design patterns is called pub-sub (for publication and subscription). In pub-sub, objects can both be message publishers and message subscribes. As a publisher, an object would, analogously, be shouting "Here is a message" from a window. As a subscriber, an object would be, again being analogous here, listening for a particular kind of shouting (say someone shouting the price per barrel of oil or shouting every time a user logs into a system).

Pub-sub is great. It's super easy, JavaScript has awesome support for event publication and subscription, and everything was great for a while. Then big data happened.

slideIn software projects that approach the complexity of an organism's genome (think Facebook or Gmail) and are worked on by thousands of different programmers, pub-sub can quickly become a nightmare. What happens if I tell one object to turn a lightbulb off every time a user logs in **slide** and a programmer in Islamabad tells another object to turn the same lightbulb *on* every time a user logs in? If a user logs in, the state of the lightbulb is no longer deterministic and is instead left to chance: which subscriber acted first and which acted second?

slide

These sorts of race conditions (where two objects race to do the same thing at the same time) was a huge problem particularly at Facebook (and apparently is why, if you can

remember back a decade or so, Facebook got really broken around the time they rolled out chat. Apparently their chat code was wreaking all kinds of havoc on the pub-sub architecture of the main client). To solve the increasing number of bugs proliferating around race conditions, Facebook developed a new design pattern they call **slide** Flux.

Flux is an "application architecture … for building client-side web applications" (*Flux*, n.p.). It works through "a unidirectional data flow" (*Flux*, n.p.). Unlike pub-sub where everyone is both a publisher and a subscriber, Flux rigorously controls who can send messages and, most importantly, where the results of these messages are processed and stored. **slides** Flux consists of action creators, a message dispatcher, and a central data store. An action is a signal to do something, just like in pub-sub. Many components can publish actions, but their only subscriber is the Dispatcher. Additionally, the Dispatcher and

the Store have an exclusive pub-sub relationship. After the Store has updated, any components that depend on state changes receive updates from their subscriptions to the store. Flux does two things: 1) it centralizes pub-sub and, most importantly, 2) it defines the only direction in which messages flow in the system: from user events, to actions, to the dispatcher, to the store, and back to views where they are displayed to the user. This is why Flux is called “unidirectional” in its documentation.

[slide]So, with the lightbulb example earlier (where there were two objects both changing a lightbulb), in Flux, the example would be much more complicated. The program would be initialized with a global state that tracks lightbulb status (“on” or “off”) and a reducer that, whenever a login action is received, sets this state to its opposite (“on” to “off” or “off” to “on”). Now, the object that actually controls the bulb will subscribe to

this global state and, when the state changes, this controller changes the lightbulb accordingly.

This has the effect of making application state a global, shared resource, rather than the properties of individual objects. Moreover, it makes it much clearer what is happening when an application is running. Since the lightbulb's state is managed in one place (globally), there is no possibility of two objects both managing the same bulb, unaware of one another, as the bulb's state is only tracked in one place.

However, it also makes turning a lightbulb on and off really complicated. All this complexity is warranted, the creators of Flux argue, because of the related complexity of application platforms on the scale of something like Facebook. Flux was created to address the fact that within Facebook's massive software development hierarchy, teams were inadvertently effecting one another's code through the creation of race

conditions like the one I described earlier. As Facebook transitioned from a place to sort out which girls at Harvard were hot and became an application platform, their developers increasingly realized that full freedom of application design was a problem.

So, what does this have to do with rhetoric?

As I suggested, I think it's important to think about platforms less as places where rhetoric happens and more as infrastructures that enable and disable certain kinds of rhetoric. As an earlier example of what I mean by this distinction, we all know that many rhetorical devices were worked out in the Athenian law courts. Thus, we could say that these law courts were similarly spaces where rhetoric happens ("behold, rhetoric"), but we would be missing the more important point: spaces condition certain rhetorics to happen within them (which we've known since at least Roxanne Mountford's "On Gender

and Rhetorical Space" (Mountford)). In the law courts, **[slide]** they used a water clock called a clepsydra (literally a "water thief") to keep time. It regulated the length of speeches by causing a fixed amount of water (that varied depending on the type of case being tried) to flow out in a fixed amount of time. Moreover, as Suzanne Young traces in "An Athenian Clepsydra," literally everyone who was anyone in ancient rhetoric at one point or another commented on time flowing like water and began to use the figure of the clepsydra to structure their arguments (Young).

What I want to show by bringing a ceramic water clock to a discussion of Facebook is that rhetorical spaces are actually rhetorical infrastructures that don't just contain rhetorical speech but structure them in an active way. **slide** In her famous essay inaugurating infrastructure studies, Susan Leigh Star writes that infrastructure does "the invisible work of creating a unity of

action in the face of a multiplicity of selves" and works through including and excluding, creating those who count and those who don't (Star 29). In looking at Facebook through the lens of its code and the architecture that code builds, I want to argue that rhetorical platform studies is at a key moment to understand how assumptions about what are "natural" features of discourse are, instead, the products of human technological choices that are now taken for granted.

slide I am also interested in thinking about Flux as a form of *detractio*, the rhetorical operation of subtraction. One of the *quadripartita ratio*—along with addition, permutation, and transposition—it's "purpose", as Erasmus writes in *De Copia*, "is to enable you so to include the essentials in the fewest possible words that nothing is lacking" (**???**, I.vi). While it's probably ironic that there is much more said about addition in the canon of rhetorical theory, especially in *De Copia* ("your

letter pleased me greatly"), subtraction is, I argue, extremely important in a digital age.

slide As Shawna Ross has excavated, *detractio* is the form of online communication solicited by social media platforms such as Facebook: these platforms prompt "forms of verbal compression that are dense, allusive, and lossless, self-contained programs that can output large quantities of information by referring to a tiny percentage of the original data" (Ross 26). We write in small, dense bursts which speak to a "relationship between the manifest digital enunciations of social media and the larger context to which outsiders are typically not privy," but why I look at Flux is because I want to ask if this is a natural product of social media platforms or if it could somehow be seen in the code itself (Ross 26).

Baking *detractio* into the software itself is an important turning point in the development of large-scale web

applications. I cannot stress enough that Flux is a real pain in the ass to learn to code. The structure of the application is overkill for simple projects; however, I first taught myself Flux when a project I was working on using pub-sub manifested one of those indescribable and unpredictable race conditions as it grew in complexity. Flux gives order but it gives it at the price of convenience. At the same time, and this where I first got interested in this project, as I conditioned my programming to Flux's structures, I began to think in terms of Flux, like in my analog life.

slide So, Flux was devised as a particular programmatic *detractio* to solve the problem of application complexity Facebook was facing as its app went huge in scale. However, Flux importantly, I argue, shaped the discourse on that platform (or was shaped by it; I'm not sure). As I mentioned earlier, Flux is a feed-forward architecture: messages flow in one direction

only; there is no means for reflection, commentary, crosstalk, or feedback.

As we know, it's hard to have a discussion on Facebook; we usually attribute this to political polarization, the need to score points through hot takes, or the reduction of complex ideas to memes. However, what if this discussion is attributable to the architecture of the code itself: messages only go one direction in Flux; messages only go in one direction on Facebook?

slide I am interested in this question for two reasons. In the essay I quoted earlier, Susan Leigh Star describes coming to consciousness regarding infrastructure and power through the experience of being allergic to onions in a McDonalds. As someone who didn't count according to the logics of standardization and recognized deviations-from-standard in that company's logic, she found she could not get them to serve her a

hamburger without onions (it was easier to just pick them off). From this, she writes:

> The power of feminist analysis is to move from the experience of being a non-user, an outcast or a castaway, to the analysis of the fact of McDonald's (and by extension, many other technologies)- and implicitly to the fact that 'it might have been otherwise,' - there is nothing necessary or inevitable about the presence of such franchises. We can bring a stranger's eye to such experiences. (Star 38)

This question of "might it have been otherwise" (which is also the question asked by reification, btw) is central to the exploration of the discourse standards Facebook encodes in its software, at the level of the software itself.

Additionally, I am working on a larger project about the rise of functional programming techniques (of which Flux is part) and the culture of risk management that shapes social media and digital culture generally. Flux is part of a move to use more heavily architected and mathematically provable software

development techniques as a way of hedging against chaos, but I worry that these software tactics are bleeding into the culture circulated by these software artifacts. For instance, as I have been arguing in this paper, Flux is a unitary direction message architecture and it creates a rhetorical space in which everyone is arguing but no one is persuading, a space without the possibility of reflection.

My larger book project is entitled *Immutability*, which takes its name from a different software feature Facebook uses extensively (in which data once set, can never be changed). Immutability, etymologically, refers to the inability to change; however, in Roman times, it also specifically referred to the inability to be persuaded by discourse. In other words, I want to trace out how, in fear of software bugs, these social media platforms are building systems that dismantle and reassemble in strange, new ways the basic building blocks of rhetoric itself.

slide As Susan Leigh Star argued, “When standards change, it is easier to see the invisible work and the invisible memberships that have anchored them in place” (Star 44). We are at a moment where the infrastructure of rhetoric is profoundly changing but without being attuned to this infrastructure, we risk missing the story it is telling us.

Thanks

Mountford, Roxanne. “On Gender and Rhetorical Space.” *Rhetoric Society Quarterly*, vol. 31, no. 1, 2001, pp. 41–71, doi:*10.1080/02773940109391194*.

Nietzsche, Friedrich. “On Truth and Lying in a Non-Moral Sense.” *The Birth of Tragedy and Other Writings*, edited by Raymond Geuss and Roland Speirs, translated by Roland Speirs, Cambridge UP, 1999, pp. 139–153.

Ross, Shawna. “Hashtags, Algorithmic Compression, and Henry James’s Late Style.” *The Henry James Review*, vol. 36, no. 1, Feb. 2015, pp. 24–44, doi:*10.1353/hjr.2015.0005*.

Srnicek, Nick. *Platform Capitalism*. Polity, 2016.

Star, Susan Leigh. “Power, Technology, and the Phenomenology of Conventions: On Being Allergic to Onions.” *Boundary Objects and Beyond: Working with Leigh Star*, edited by Geoffrey C. Bowker et al., MIT UP, 2016, pp. 263–289.

Young, Suzanne. “An Athenian Clepsydra.” *Hesperia: The Journal of the American School of Classical Studies at Athens*, vol. 8, no. 3, 1939, pp. 274–284.