Original Research Article

Big Data & Society
July–December 2016: 1–12
© The Author(s) 2016
Reprints and permissions:
sagepub.com/journalsPermissions.nav
DOI: 10.1177/2053951716642456
bds.sagepub.com

Stacked spaces: Mapping digital infrastructures

Till Straube

Abstract

This article turns towards the spatial life of 'digital infrastructures', i.e. code, protocols, standards, and data formats that are hidden from view in everyday applications of computational technologies. It does so by drawing on the version control system Git as a case study, and telling the story of its initial development in order to reconstruct the circumstances and technical considerations surrounding its conception. This account engages with computational infrastructures on their own terms by adopting the figure of the 'stack' to frame a technically informed analysis, and exploring its implications for a different kind of geographic inquiry. Drawing on topology as employed by Law and Mol, attention is given to the multiplicity of spatialities and temporalities enrolled in digital infrastructures in general, and Git specifically. Along the lines of the case study and by reading it against other literatures, this notion of topology is expanded to include the material performation of fundamentally arbitrary, more-than-human topologies, as well as their nested articulation, translation and negotiation within digital infrastructures.

Keywords

Git, infrastructure, stack, technology, topology, version control

Introduction

Critical analyses of information and communication technologies (ICT) necessarily focus on specific aspects of a wide-ranging, complex subject matter. In human geography, the entry point to picking apart the various workings and implications of digital technologies is their enigmatic relationship to physical space, and various influential works have proposed different analytical framings to this problem (Aoyama and Sheppard, 2003; Batty, 1997; Graham, 2005; Kitchin and Dodge, 2011; Thrift and French, 2002; Zook and Graham, 2007). Empirical studies tend to center around certain focal points where technological innovation interfaces with social spaces, such as the 'smart city' (Gabrys, 2014; Halpern et al., 2013; Kitchin, 2015; Shelton et al., 2015), augmented mobilities (White, 2016; Wilson, 2012), or the digital divide (Graham, 2011; van Dijk, 2006). Geographic research often focuses on the effects that specific applications and ICT in general have on society and space, but it tends to be less concerned with the technical details of how digital technologies do work to effect these changes.

The present text looks behind the representational interfaces of the applications operated by the user on their computer or mobile device, and focuses on the 'infrastructural' technologies that run in the background and link them to the hard materiality of bitwise operations, magnetic storage, and optical cables. There are a myriad of computational processes, languages, protocols, formats, and standards involved in the functioning of any computational system, and the large majority of their workings escape understanding by any single computer engineer, let alone the end user.

A key challenge to studying protocols, formats, and standards is that making them visible requires work. Infrastructural technologies can be reclaimed from the background through studying their historically situated

Department of Human Geography, Goethe University Frankfurt, Frankfurt, Germany

Corresponding author:
Till Straube, Department of Human Geography, Goethe University Frankfurt, Theodor-W.-Adorno-Platz 6, 60629 Frankfurt am Main, Germany.
Email: straube@geo.uni-frankfurt.de

Creative Commons NonCommercial-NoDerivs CC-BY-NC-ND: This article is distributed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 License (http://www.creativecommons.org/licenses/by-nc-nd/3.0/) which permits non-commercial use, reproduction and distribution of the work as published without adaptation or alteration, without further permission provided the original work is attributed as specified on the SAGE and Open Access pages (https://us.sagepub.com/en-us/nam/open-access-at-sage).

assemblage, i.e. by drawing on archives to ''artificially produce (...) the state of crisis'' that led to their conception and proliferation (Latour, 2005: 81). When Manovich recounts the prototyping of the personal computer by Alan Kay and his team at Xerox PARC in the 1970s, he aims to ''look into the thinking of the inventors of cultural computing themselves'' by asking: ''What were their reasons for doing this? What was their thinking?'' (Manovich, 2013: 56, 60; emphasis omitted). The result of that inquiry is an informative dissection of the contingent design choices and resulting affordances that continue to define the fundamental formats, mechanisms, and user interfaces of a wide range of present-day multimedia. It is through retracing the specific circumstances and considerations which have led to transitions and disruptions in fundamental digital technologies, then, that these systems come into view as ''matters of concern'', rather than ''matters of fact'' (Latour, 2004: 225).

The object of inquiry chosen here is a version control system, or source control management tool (SCM), by the name of Git. Version control is a technology that tracks changes made to the source code of a piece of software. It is also employed to facilitate cooperation on software projects, as SCMs provide for mechanisms that prevent developers from accidentally overwriting each other's changes, and make visible who supplied which revisions to the codebase. SCMs are of interest to this exercise of unpacking digital infrastructures because they are largely hidden from view. Most end users of digital technologies never come into direct contact with revision control systems, although any given application that they do use, from on-line shopping sites to navigation systems, was almost certainly developed with the help of version control technology. What is more, SCMs typically do operations on code as textual data, thus blurring the preconceived dividing line between code and data, and speaking to the complexity and fluidity of digital infrastructures (Mackenzie, 2016).

Software engineers routinely use revision control systems to track and share changes without thinking twice about their mechanisms, or where they came from. Thus, even to most people who *are* familiar with SCMs and engage with them on an everyday basis, version control appears as an already-established technology that can be employed and built upon in different ways, but is rarely subjected to scrutiny itself. In this way, Git is similar to technologies like the Simple Mail Transfer Protocol (SMTP, the standard behind e-mail), or the Global Positioning System (GPS). It is only for a small number of influencers that these 'basic' technologies are not beyond second-guessing, modifying, and re-engineering. For the rest of us, SCMs are an unexciting, technical necessity for collaboration on code. While version control systems may inform workflow and its results more than their users realize, the technology itself remains in the background: the less it is noticed, and the less time is spent actively thinking about version control, the better. In this infrastructural quality and relative obscurity lies my interest in excavating the technologies involved in version control.

Of a variety of possible candidate SCMs, Git was chosen here for several reasons. First, it is open source software, and as such it is possible to subject Git to technical scrutiny at a level of detail that would be impossible with pieces of software that hide their source code. Secondly, while there are no reliable usage statistics of version control systems, it is nonetheless apparent that Git has in recent years experienced the most striking surge in application,[1] and through the proliferation of the web-based repository hosting service GitHub it is now being employed to collaborate on a wide range of textual data (including legal documents, vector images, cookbooks, or DNA information). Finally, Git is a rather recent addition to the family of revision control software, and, paired with the considerable interest it has generated within the software development community, this has led to a relative wealth of publicly accessible information pertaining to its initial development.

The goal of my analysis of Git as digital infrastructure is two-fold: First, it is an exercise in empiricist engagement with computational technologies. Taking methodological cues from science and technology studies and actor-network theory literatures, I offer an account of the workings of infrastructural technologies that is sensitive to the technical considerations surrounding their design by computer engineers. Following Star's (1999: 377) ''call to study boring things'' as well as Ong and Collier's (2005: 10) notion of ''technical criticism'', this approach pursues a situated understanding of digital devices through careful unpacking of a multitude of interfaced systems, languages, protocols, standards, and formats.

The second and related objective of the present text is to irritate preconceived notions of space from an interdisciplinary position that takes seriously both technical considerations and abstractions deployed in computer science, as well as social science literature examining (digital) technologies. What spatialities and temporalities emerge when an infrastructural technology is unpacked and its inner workings are made visible, and what are suitable models and concepts to think about the technological interdependencies and interfaces contained within digital infrastructures in spatio-temporal terms?

In what follows, I first reconstruct the circumstances, events, and considerations surrounding the development of the version control system Git. This was achieved by closely analyzing on-line documents including articles, blog posts, presentations, published interviews, and messages to public mailing lists. As a stylistic device, I present my findings in a story-like passage that brings back to life the moment of Git's construction and lets its protagonists speak without an a priori imposition of any interpretational framework. Next, I distill from this account the model of the stack as a natively technical framework to think conceptually about the various layers of protocols, code, and data formats involved in the functioning of an infrastructural technology like Git. Finally, I develop a notion of topology chiefly inspired by science and technology studies to more thoroughly theorize the way in which multiple spatialities and temporalities are implicated in stack-like infrastructural systems.

Kernel trouble

This section of the article recounts the initial development of Git as told by the actors immediately involved, the central figure being lead developer Linus Torvalds. My account refrains from sourcing pieces of information individually for stylistic purposes. References to the archive materials (which were coded thematically and combined to a coherent narrative) are instead listed collectively in the appendix.

In the beginning of April 2005, Linus Torvalds (the principal author and eponym of the Linux operating system) had a problem to solve. For years now, the core developers of Linux had relied on the version control tool BitKeeper to manage the many changes to the Linux 'kernel' (the operating system's vital codebase) which were needed to continuously implement new requirements and to smooth out bugs. Use of this SCM was optional for Linux developers; some contributors still preferred to share code by e-mailing standardized files containing changes (so-called 'patches') directly, as in the early days of Linux development. But Torvalds and other key contributors used BitKeeper to keep track of these changes and to resolve conflicts between versions that had revisions made to the same part of the code, making it the de facto standard for the cooperative development effort.

Torvalds had not much cared that, unlike Linux, BitKeeper was proprietary software. The CEO of the company that owned BitKeeper, Larry McVoy, was himself a contributor to the Linux kernel and had provided a 'community license' free of charge for open source projects like Linux. Additionally, the BitKeeper client software allowed for limited interoperation with other version control protocols. As far as Torvalds was concerned, BitKeeper simply got the job done. But other kernel contributors had been more concerned by the software's proprietary status. They had been skeptical of a 'black-boxed' commercial tool managing the Linux codebase, and thought that this was contrary to the open source philosophy of the Linux project. There had been occasional 'flamewars' on the Linux kernel mailing list about this issue. Things had finally come to a head when a veteran Linux developer reverse-engineered the BitKeeper protocol (in order to read metadata about code revisions from communication with the BitKeeper servers, rather than merely downloading the most current version of the code). For McVoy, this had been the first step to building a powerful unlicensed client, and a threat to his business. In reaction, he had revoked the community license, and now the Linux community was facing going back to e-mailing patches and 'tarballs' (compressed archives).

Until now, Torvalds' experience with SCMs had been strictly from a user perspective. While some sort of version control was a necessity when handling a large cooperative project, SCMs were thoroughly unexciting to him from a technical standpoint. But when researching open source alternatives to BitKeeper he could not find a solution that would work for the Linux community. From Torvalds' perspective, most SCMs (like the de facto industry standard CVS or Apache's Subversion) had a flawed approach to revision control, resulting in a 'broken' model: they relied on a central repository that would keep track of the changes made by all users. In a centralized system, users would usually 'check out' files, make their edits to the code, and check the files back in. Testing would have to be centrally coordinated, and experimental 'branches' (alternative, concurrent versions) of the source code would result in clutter on the server. Most importantly, a centralized system would invariably give rise to the political question of access: Which developers would have the right to upload changes to the kernel code? Grant privileges to too many contributors, and there would be bad 'commits' (batches of code revisions). Choose too few, and those with access would spend all their time working in changes sent to them from those without commit privileges. Centralized SCMs might work (though badly, in Torvalds' opinion) for a tightly managed company environment, but not at all for the loose network of contributors to the Linux kernel.

In contrast, BitKeeper had been accommodating a decentralized approach to version control. Rather than working on single files stored on a central server, contributors each had their own copy of the entire codebase, and were able to make changes and run tests locally. Experimental features could be tried out without compromising anyone else's version of the code. Once a contributor felt confident that his[2] revisions were an improvement to the code, he would share them among the community, and everybody could choose for themselves whether to incorporate the changes or not. While everybody was looking at Torvalds' copy of the kernel as the 'official' version, in theory nothing prevented any other developer from 'forking' (i.e. creating an independent version of) the project and taking development of the operating system into another direction. What changes ended up in the kernel release was determined by 'networks of trust' rather than bureaucratic administration of privileges. What is more, in a decentralized environment backups were not an issue, because so many versions of the same codebase existed 'out there.' Torvalds felt that this model was the best fit for open source development and, more importantly, for his own personal workflow. Decentralized SCM was the way to go, and Torvalds feared that if they had to switch to a centralized tool in a pinch, it would be very hard to ever go back.

To be sure, there were a few decentralized SCMs available that were free and open source software. The biggest problem with these tools, however, was performance. One of the more promising candidates, a version control tool called Monotone, took over two hours just to import the Linux kernel's 17,291 files at the time. Importantly, Torvalds recognized that performance was not simply a question of doing the same thing faster or slower. If 'diffing'—i.e. comparing two versions of a (partial) codebase line by line and highlighting the changes—was slow and resulted in a 'get-another-coffee-moment' every time, developers were simply less likely to diff at all. This would result in redundant work and bad code. Similarly, if one had to plan ahead and set aside a day for 'merging' (combining) two or more concurrent branches, this would discourage contributors from opening their own experimental branches in the first place, and result in clutter and bad commits.

Despite its "glacial" speed, however, there was one thing in particular that Torvalds found intriguing about Monotone: the way in which it hashed[3] objects. In lieu of BitKeeper, Torvalds had already started keeping a record of consecutive patches with their respective hash codes to be able to quickly verify the content of code revisions and recognize corrupted data. But Monotone not only used a superior hash algorithm (SHA-1 rather than the MD5 checksums Torvalds had adopted from BitKeeper) –the Monotone developers had also incorporated hashing much more rigorously, to the point where every version of a file or a directory, or indeed of the entire project, could be efficiently referenced by its SHA-1 hash, rather than by file names, version numbers, timestamps and so on. This approach immediately appealed to Torvalds, even if it meant abandoning his own series-of-subsequent-patches model.

Torvalds realized that writing his own version control tool would not be much of a challenge if he only got the basic ideas right. As it happened, the latest stable Linux kernel (version 2.6.11) had just been released in March, and he had some time to spare before things would heat up again in the next release cycle. *Give me two weeks,* Torvalds thought, *and I will write a better SCM than anything out there right now.* He went to work, and after a few days Torvalds' revision control project became self-hosting, with Torvalds' first commit message reading: "Initial revision of 'Git', the information manager from hell." This commit marks the creation of Git (the name deriving from a British slang term for a rotten person—the second project Torvalds named after himself, he jokes). Two months later, the custom SCM was employed to manage the release of Linux 2.6.12.

Initially narrowly designed around what he wanted his own workflow to be like, Torvalds' solution to version control was notoriously difficult to use: As another Linux lead developer remarked, Git seemed to be "expressly designed to make you feel less intelligent than you thought you were." Still, Linux developers and other open source communities quickly recognized Git's technical sophistication and practical potential, started to contribute to the new project, and employed it to collaborate on their own codebases.

Infrastructural stacks

Unlike the fundamental technological changes brought into the world by Alan Kay and his team when introducing the personal computer, it is hard to pinpoint any innovation per se that Torvalds might have contributed by authoring Git. Like virtually all software, the Git project was built on a variety of existing technologies. For example, it implements the SHA-1 hash function, it provides a Secure Shell (SSH) interface for encrypted network connections (next to its own SSH-like protocol), and it was written in C, a general-purpose programming language developed in the early 1970s. Git additionally employs various code 'libraries' which are used in a wide range of open source software to avoid defining the same basic functions over and over again: At the time of this writing, installation of Git on a Linux system requires the libraries 'curl' (for data transfer), 'zlib' (for compression), 'openssl' (for encrypted remote connections), 'expat' (for parsing XML), and 'libiconv' (for character encoding). What is more, Torvalds' design choices were all inspired by existing SCMs—he cites BitBucket, Monotone, and his own history-of-patches model as Git's "three parents" (Torvalds, 2006d). In the spirit of open source culture, Git can be thought of as a "remix" (Lessig, 2008) of existing tools and technologies.

Not only is Git inspired by and built upon various technologies that are external to the project itself. With its focus on flexibility, performance, and structural integrity (rather than an intuitive user interface), Git itself comes into view not so much as an end-user application but as a *format* for storing versions of files, a *protocol* for exchanging code, and a *standard* for distributed development processes—a ''form for handling forms'' (Easterling, 2012: Chapter 6, para. 5). Like SMTP or GPS, Git recedes into the background so easily because it is an *infrastructural technology*.

When it comes to interactions and dependencies between digital system components like protocols, data formats, or software, computer engineers tend to conceptualize them in terms of hierarchically organized 'stacks' (Solomon, 2013). The idea of complementing protocols that are 'layered' on top of each other dates back to the Open Systems Interconnection (OSI) model developed in the late 1970s. This conceptual framing of communication functions within computer networks was the result of parallel efforts by the International Organization for Standardization (ISO) and the International Telegraph and Telephone Consultative Committee (CCITT) to provide a standard reference that would enable communication between a diverse set of parallel (and often proprietary and competing) networking technologies (Zimmermann, 1980). The model articulates distinct functions that have to be fulfilled by networking protocols in order for two computers to communicate, resulting in seven abstract, hierarchical layers.

It is important to note that protocols at different layers of the OSI model handle the same base information simultaneously, but at different levels of abstraction. For example, the hypertext transfer protocol (HTTP) handles the source code of a web page as textual data within the application layer. The same raw data is handled as 'packets' within the network layer, and as a simple consecutive string of bits at the physical layer. Generally speaking, any given networking protocol can be thought of as implementing a specific layer, and must provide an interface only to the two specific protocols that locally implement the layers directly above and below. By adhering to this standard, any communication between two computers can be translated from the 'topmost' application layer—e.g., a web page in HTTP—through a series of interfaces all the way 'down' to the physical layer (i.e. protocols that handle the actual signaling through electrical currents, optical pulses, sound or radio waves) and back 'up' again to be decoded as needed—*independently* of any specific protocol.

As a global frame of reference for communication protocols, the OSI model enabled the construction of the Internet as a ''network of networks'' (Braden, 1989a: 7). A protocol 'stack', then, is the local implementation of a suite of protocols that spans several conceptual layers. The best known example of a protocol stack is the Internet protocol suite (often referred to as TCP/IP for its first and most vital components) which has become the de facto standard for the Internet as we know it (Braden, 1989a, 1989b).

But 'stack thinking' extends beyond the world of networking protocols. Web servers, for example, are also thought of as stacks of complementary, interdependent software. The open source 'LAMP stack' consists of Linux (as operating system), Apache (as HTTP server), MySQL (as relational database management system), and PHP[4] (for server-side scripting of dynamic web pages). Here again, the four pieces of software serve different but interdependent functions. They are organized hierarchically in that a series of 'vertical' translations through the stack can be traced with each request from a remote agent: in order to serve a dynamic web page, Apache calls a PHP script (through its PHP interface, or 'module'), which in turn connects to the database (through PHP's MySQL 'extension') to dynamically generate the web page. All three depend on the Linux operating system and its various components to handle the details of how data is read from the hard disk, how the processor does calculations, how data is received and sent over the network, and so on.

There are further instances of the stack as a conceptual model for organizing processes and data in computer science, including the 'call stack' that represents layered subroutines of a computer program, or the stack as an abstract data type in some programming languages (Solomon, 2013). But for the remainder of this section the focus will be on the ways in which the stack as a conceptual model has traveled beyond the realm of communication standardization and technical specifications, and begun to serve in media studies and the social sciences as a tool for critical analysis of ICT.

Bratton applies the model of the stack to computing at large: as ''a machine that serves as a schema, as much as it is a schema of machines'' (Bratton, 2015: para. 1). He positions the stack as a contemporary nomos—a productive, geo-political metastructure organizing planetary space and society. At the same time, the stack is Bratton's tool for dissecting a vertically integrated, dynamic, heterogeneous assemblage organized in various systems that ''swap phase states,'' and continuously territorialize and de-territorialize the same components ''indecipherably'' (Bratton, 2012: section 3, para. 2). This reading of the stack as an elusive, exceedingly complex meta-architecture allows Bratton to include various imaginative 'layers' of the stack into his sweeping analysis (including the cloud, ubiquitous computing, augmented reality, the user, or the interface) while maintaining claims to a planetary perspective.

As part of a larger project to re-frame current digital technologies within speculative post-capitalist contexts, Terranova (2014) draws on Bratton's conceptualization of the stack in order to urge appropriation of a series of emerging digital technologies for their emancipatory potential. She proposes working towards the materialization of a "red stack" by "engaging with (at least) three levels of socio-technical innovation: virtual money, social networks, and bio-hypermedia"—the latter signifying the increasingly close integration of bodies, devices, and applications (Terranova, 2014: 390). By imagining subversive uses of these 'layers', Terranova suggests, the technologies in question cease to be mere tools of capitalism, and can contribute to alternative socio-economic configurations. Similarly to Bratton, she employs the stack as a framework that inspires a multi-layered, simultaneous consideration of integrated technological systems in broad strokes. This reading is far removed from the stack's original technical context concerned with standardization, hierarchies of dependency, translation of data formats between protocols, degrees of abstraction, and so on. Latour provides the reverse perspective from the software layer when he asserts that mobile phones are "composed of writings all the way down" (Latour, 2008: 4).

In contrast, Mattern's analysis of smart cities and their various interfaces makes use of the stack in a way that incorporates these technological considerations. She draws on Solomon's (2013) insightful genealogy of the stack and stack thinking in computer science in order to articulate a model of the "urban stack" for a problematization of transparency and citizenship in smart city contexts (Mattern, 2014: section 2, para. 3).

White (2016) also proposes a model of the stack that stays closer to its technical origins. He engages with the works of Benkler (2006) and Zittrain (2008) to draw from their analyses models of ICT as composed of various conceptual layers. White goes on to articulate a broad model of the stack—composed here of hardware, software, data(base), and interface—that serves as a heuristic device in his critical analysis of a specific technology: the ordering of taxis by smartphone. Kitchin (2016) similarly organizes a wide range of elements according to a stack-like hierarchical logic deemed useful for systematic analysis of concrete technologies.

While the above articulations of the stack differ in the specific layers that they identify, it is nonetheless possible to distill from them a general working notion of the stack as an analytical tool for 'slicing' complex systems along several conceptual lines, exposing cross-sections that allow different views on the same system, each with their own way of abstracting digital information, and each contributing to the functioning of the technology. Importantly, the stack is not simply an enumeration of different elements that constitute a whole. Instead, each of its layers is an articulation of a specific logic and already encompasses the entire system. It is impossible to 'add' software to hardware, or data to code—they each exist on separate conceptual planes and are, in themselves, lacking nothing: it is only from the perspective of the hardware layer that Kittler (1995) could famously claim that "[t]here is no software."

At the same time, the stack clearly establishes hierarchy: each layer depends on the one below to function, and adds a dimension of abstraction that is in turn the base for the layer above. The stack, then, is not hopelessly indeterminate or indecipherable, as Bratton seems to suggest. The hierarchy of layers is real; it is how computational systems are purposely built, and for each individual action on the 'surface' of the stack it is possible to trace a gradual translation, a transparent chain of deciphering calls through a series of descending levels of abstraction all the way 'down' to the material handling of bits.

This notion of the stack adheres to the basic principles of its original use as a technical model in computer science. And yet, as a frame of reference it allows for analysis of systems more generally than the OSI model which is concerned only with specific interdependencies of networking protocols. Unlike Bratton's metaphorical use of the stack, it is a tool for dissecting specific technologies, and stops short of explaining computing at large, or articulating a new social order. The stack as envisioned here is an analytical tool that takes a real technical model (actually informing system building practices by computer engineers), and slightly widens its scope while staying close to its original context: its application is metonymic rather than metaphoric. In this sense, the stack can be understood as a "middle-range theory"; one that goes beyond mere description, but does not undertake the effort to explain the world (all systems, or all of society) in a uniform manner (Merton, 1968; Wyatt and Balmer, 2007).

Coded topologies

In order to describe a technological stack and the ways in which it negotiates the various tensions outlined above, a multiplicity of concurrent logical strata have to be considered. In the vocabulary of Deleuze and Guattari (1987: 12), each layer can be understood as a "tracing", and the stack is the "map" containing a multitude of iterative tracings along different lines of flight.

Software engineers routinely work with schematic visualizations when designing systems and interfaces. What is a database schema or a file system tree but a (spatial) tracing of a specific cross-section of the stack? I contend that the various articulations of relational systems making up the layers found within digital infrastructures should be taken seriously as spaces proper—if anything, they are *more* relevant to understanding the technological stack than Euclidian space (or, for that matter, other spaces 'mapped onto' the system by social science researchers).

Situating the stack thus requires a strategy for dealing with a multiplicity of time-spaces, and I propose that a certain understanding of topology can underpin such an endeavor conceptually. As Martin and Secor (2014: 2) note, however, there is a "dizzying diversity" of geographic texts employing vastly different concepts of topology in their analyses. This ambiguity of the term persists where it is applied to digital technologies. Marres (2012), for example, uses topology synonymously to a Latourian symmetry in socio-technical arrangements in her analysis of smart meters, and Rogers (2012) employs topology to recount a series of specific historical geometries of the Internet.

In geography, topology is most commonly understood as an antithesis to topography. In this reading, topography is concerned with producing images of the earth's surface, measuring distances, and locating objects in Euclidian space. Topology, on the other hand, is only concerned with qualities of connectivity, and has no use for concepts of scale or distance. A hiking map is said to be topographical, while a subway map is topological. Where topography follows the logic of territory, topology follows that of the network (Amin, 2004; Opitz and Tellmann, 2012). This understanding of topology is sometimes illustrated by Serres' metaphor of the crumpled handkerchief (Serres and Latour, 1995; Murdoch, 1998) or Euler's solution to the problem of the seven bridges of Königsberg (Shields, 2012). Topology-as-opposed-to-topography has been employed by various authors in an effort to incorporate actor-network theory into geography (Allen, 2011; Latham, 2002, 2011; McFarlane, 2009; Murdoch, 1998). However, such an understanding of the term has been criticized for oversimplifying or even misreading the mathematical field of topology (Sha, 2012). Crucially, topological spaces *can* include measures of distance, or metrics: "the point is to understand Euclidean space as *one possible topology among others*" (Martin and Secor, 2014: 11; emphasis in original).

An alternative approach that explicitly positions Euclidian space (and linear time) among a multiplicity of alternative time-spaces is the work of Law and Mol, who propose a reading of objects as effects between multiple topologies. This ontological assertion is most clearly articulated when the authors interpret Latour's figure of the immutable mobile along those lines:

> [W]e find that the immutable mobile achieves its character by virtue of *participation in two spaces:* it participates in *both network and Euclidean space*. And such is Latour's trick. To talk of an 'immutable mobile' is to elide the two. (Law and Mol, 2001: 612, emphasis in original)

For Law and Mol, then, topology is a device to examine an object by deploying the spaces that it enrolls. Following this approach, alternative topologies such as 'network', 'fluid', 'fire', or 'gel spaces'—each with their own logics of proximity and distance, change and continuity, absence and presence—have been deployed to shed light on the hidden topological life of a diverse set of objects, such as colonial vessels, a water pump, the aerodynamic design of a fighter jet, or alcoholic liver disease (Law, 2002; Law and Mol, 2001; Law and Singleton, 2005; Sheller, 2004).

This notion of topology resonates strongly with the figure of the stack. Just as its layers are distinct complementing views of an entire system, these concurrent time-spaces can be thought of as complete tracings of objects from different topological perspectives, and the analytical focus is drawn to the translations and dis/continuities at the interfaces of these relational systems. In what follows, I will read this approach against the Git case study as well as additional literatures in order to draw out some implications of topological thinking for studying infrastructural technologies, and address several problems arising from its application in this context.

The first issue is with the suitability of preconceived topologies (such as 'fluid' or 'fire' spaces) for the examination of digital infrastructures. Rather than imposing spatial metaphors *onto* digital technologies, the goal should be to engage with and draw out specific spatial articulations encountered *within* the various inscriptions of digital devices. The concept of autospatialization as employed by Lury et al. in their topological analysis of automated sorting, comparing, and calculating practices can lead to such an understanding of technologically performed spatialities (Lury et al., 2012; Parisi, 2013; Ruppert, 2012). In the mathematic understanding of topology, there is a fundamental arbitrariness to what spaces can be articulated and examined in regard to their properties. Rather than highlighting characteristics of digital technologies to fit preconceived metaphoric spaces, the approach proposed here draws on their very overflowing of existing frames of reference in order to arrive at informative descriptions of the multiple, concurrent spatialities at work in the stack.

Git negotiates a variety of spatialities that are built into its infrastructural workings, including hierarchical file systems, networks of developers (often referred to as 'networks of trust'), the "text space" of code (Chun, 2008: 161), and its own tree-like logic of consecutive commits, branches, and forks. Furthermore, Git modulates temporality: It is employed to roll back to older (working) versions of a buggy software, and because individual commits represent only the incremental difference between two versions of individual lines of code, old commits can be removed or applied to newer branches, or entire branches can be 'rebased' to newer commits. Git allows for flexible re-ordering of a consecutive, incremental timeline of modifications—nothing less than the changing of a code's history. In other words, a multitude of spatio-temporal relational systems that are folded into Git can be read directly from the object, and none of them adhere to any preconceived notion of fluid, fire, hybrid, cyber- or other space.

A second and related problem is the distinction of the spatialities in and of the stack from mere metaphor. Following Callon (2007, 2009), I assert that conceptual, diagrammatic articulations of spatiality (such as a computer network, or a database schema) are inscribed into the material workings of digital devices and become active in their performation. Drawing on experiments in quantum physics, Barad develops the notion of "spacetimemattering" to point to the inextricability of spatio-temporal frames of reference and their "dis/continuities" from those material arrangements that are employed in their observation (Barad, 2010: 244). The time-spaces of digital infrastructures, then, are not interpretational devices that are introduced in a moment of abstract reading, but rather built into the very materiality of stack-like configurations—they are *performative of* and *performed by* the devices in question.

To further complicate matters, nothing warrants the expectation that the time-spaces encountered while unpacking the stack be commensurable to human experience. For example, it is not at all out of the ordinary to employ more-than-three-dimensional calculative spaces, or nested loops and recursions in software programming. It is thus imperative to be prepared for encounters with time-spaces that are difficult or impossible to relate to, without reducing them to some more intuitive framing (just as mathematics is equipped to handle topologies well beyond human capacity for imagination). Literatures surrounding feminist materialism and critical readings of cybernetics have a history of engaging with problems of agency and subjectivity in post- or more-than-human ontologies (Barad, 2003; Haraway, 1987; Hayles, 1999; Pickering, 2002). These conceptualizations underline the necessity to widen a topological perspective to include notions of more-than-human time-spaces. Notably, efforts to incorporate such readings into media theory have pointed out the performative quality of a multiplicity of materialisms (Drucker, 2013; Parikka, 2012).

In this reading, Git's capability to handle non-linear temporalities (e.g., by 're-basing' an older branch onto the current 'head') is not simply a handy functionality for software engineers cooperating on a project. Instead, the SCM's code is understood here as enacting a 'queer' time-space that is at odds with immediate human experience, but which by virtue of translation through the stack is firmly rooted in the materiality of physical hardware. This performation of dis/continuous relational branches can be considered an example of the 'spacetimemattering' effects of digital infrastructure.

The third issue concerns the relation of time-spaces to each other. In their "Topological Manifesto", researchers at the University of Technology in Darmstadt identify a series of typical 'spaces' structured by digital technologies, such as security spaces, administrative spaces, transport and mobility spaces, or storage spaces, each with their own relational logic (Graduiertenkolleg Topologie der Technik, 2015). The question of how these spaces relate to each other is missing from that conceptual framing altogether. In Law and Mol's topological approach, objects may be enrolled in multiple spatialities and thus function as a point of connection and translation between topologies. However, the origin of these spaces remains unclear; their articulation seems to be the privilege of the researcher. Taking cues from the layer model of the technological stack, I understand the time-spaces of digital infrastructures instead as inherently connected and articulated within another. As Lefebvre notes, "[w]e are (...) confronted by an indefinite multitude of spaces, each one piled upon, or perhaps contained within, the next" (Lefebvre, 1991: 8). Serres similarly approaches topology from a position of the in-between, and understands distinct modes of temporality as different arrangements of the same coherent fabric and connected through diffuse passageways (Connor, 2004; Serres and Latour, 1995). Rather than envisioning a multiplicity of spaces as unproblematically existing in parallel, the mode of inquiry proposed here is concerned with the 'pivot points' that translate between technological time-spaces, and with their nested articulations.

Within Git, concurrent time-spaces, such as the unordered, atemporal references to snapshots of files and file trees as hash codes on the one hand, and consecutive revisions on the other, are being continuously negotiated. This interface is articulated in code—specifically the programming language C—that comes with its own (textual, spatial) syntax, rules for 'name spaces', logical loops and recursions, and as compiled computer code can in turn be located within the operating system's folder hierarchy, and so on.

In summary, the topologies that come into view in the process of unpacking the technological stack conform to Law and Mol's framing in that they can be thought of as a multiplicity of time-spaces that are necessarily enrolled in the workings of digital devices. However, they differ in that they are fundamentally arbitrary and can be read directly from the object's inscriptions, in that they materially perform more-than-human spatialities and temporalities, and in that they are nested and articulated within each other.

Conclusion

The case study of the versioning system Git served as an exercise to make visible an infrastructural technology through reconstruction of the circumstances, events, and considerations surrounding its initial design. Its 'remixing' of existing technologies and models, its focus on performance rather than intuitive usage, as well as its affordances as a format and protocol for storing and exchanging code steered this account towards the conceptualization of infrastructural systems as layered stacks, a model immediately derived from technical design practices.

Whereas critical analyses of ICT often reduce complex systems to singular figurations (such as 'big data', or 'the algorithm'), the model of the stack affords multiple perspectives on a given system. Rather than components adding up to any 'whole', each layer accounts for the system's entirety according to the layer's own logic. A situated unpacking of the stack can trace abstract actions through a series of vertical translations and reveal the hierarchical technological layers at work in performing the system.

Mapping digital infrastructures means accounting for a multiplicity of interfacing time-spaces, and the stack is a suitable framework for such topological analysis. Drawing on ontological assertions about the multiplicity of time-spaces and their enrolment in objects, the technical model of the stack can be employed to explore the material performation of fundamentally arbitrary, more-than-human topologies and their nested articulation within digital infrastructures. Such an account takes seriously the schematic models inscribed into technical devices and in terms of veracity situates them alongside more conventional notions of space and time. Within this framework, moments of articulation, translation, mediation, and negotiation take center stage, thereby positioning code, protocols, formats, and interfaces as topological operators. Unpacking digital infrastructures in such a way transcends notions of space that conventionally underpin geographic research, and opens up new perspectives on the spatial workings of digital devices.

Declaration of conflicting interests

The author declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Funding

The author received no financial support for the research, authorship, and/or publication of this article.

Notes

1. In 2014, 33.3% of respondents to a survey among users of the integrated development environment Eclipse identified Git as their primary SCM, the value surpassing that of Subversion (30.7%) for the first time in the yearly survey and revealing Git as the most popular version control tool among Eclipse users (Eclipse Foundation, 2014).
2. Other pronouns would only gloss over the stark gender imbalance in projects like Linux. A survey revealed "that women do not play a role in the development of Open Source and Free Software; only 1.1% of [the survey's] sample is female", much less than in proprietary software (Ghosh et al., 2002: 8). A subsequent study found that women are actively excluded by hostile "cultural and social arrangements" within open source communities (Nafus et al., 2006: 5).
3. Simply put, hashing is a cryptographic method that generates a fixed-length code (or hash) from an arbitrarily large set of data. Even slight changes in the input will result in an entirely dissimilar hash. Good hash functions are fast, make collisions (i.e. different inputs producing an identical hash) exceedingly unlikely, and are impossible to reverse-engineer.
4. Originally, PHP stood for "Personal Home Page". Today, the PHP manual states that the name is a 'recursive' acronym, signifying "PHP: Hypertext Preprocessor".

References

Allen J (2011) Topological twists: Power's shifting geographies. *Dialogues in Human Geography* 1(3): 283–298.

Amin A (2004) Regions unbound: Towards a new politics of place. *Geografiska Annaler: Series B, Human Geography* 86(1): 33–44.

Aoyama Y and Sheppard E (2003) The dialectics of geographic and virtual space. *Environment and Planning A* 35(7): 1151–1156.

Barad K (2003) Posthumanist performativity: Toward an understanding of how matter comes to matter. *Signs: Journal of Women in Culture and Society* 28(3): 801–831.

Barad K (2010) Quantum entanglements and hauntological relations of inheritance: Dis/continuities, spacetime enfoldings, and justice-to-come. *Derrida Today* 3(2): 240–268.

Batty M (1997) Virtual geography. *Futures* 29(4–5): 337–352.

Benkler Y (2006) *The Wealth of Networks: How Social Production Transforms Markets and Freedom*. New Haven, CT: Yale University Press.

Braden R (1989a) *Requirements for Internet Hosts: Communication Layers*. RFC 1122. Available at: https://tools.ietf.org/html/rfc1122 (accessed 18 August 2015).

Braden R (1989b) *Requirements for Internet Hosts: Application and Support*. RFC 1123. Available at: https://tools.ietf.org/html/rfc1123 (accessed 18 August 2015).

Bratton BH (2012) *On the Nomos of the Cloud: The Stack, Deep Address, Integral Geography*. Available at: http://bratton.info/projects/talks/on-the-nomos-of-the-cloud-the-stack-deep-address-integral-geography/ (accessed 7 August 2015).

Bratton BH (2015) *The Black Stack*. Available at: http://bratton.info/projects/texts/the-black-stack/ (accessed 7 August 2015).

Callon M (2007) What does it mean to say that economics is performative? In: MacKenzie D, Muniesa F and Siu L (eds) *Do Economists Make Markets?: On the Performativity of Economics* Princeton, NJ: Princeton University Press, pp. 311–357.

Callon M (2009) Elaborating the notion of performativity. *Le Libellio d'Aegis* 5(1): 18–29.

Chun WH (2008) The enduring ephemeral, or the future is a memory. *Critical Inquiry* 35(1): 148–171.

Connor S (2004) Topologies: Michel Serres and the shapes of thought. *Anglistik* 15(1): 105–117.

Deleuze G and Guattari F (1987) *A Thousand Plateaus: Capitalism and Schizophrenia*. Minneapolis: University of Minnesota Press.

Drucker J (2013) Performative materiality and theoretical approaches to interface. *Digital Humanities Quarterly*. Available at: http://www.digitalhumanities.org/dhq/vol/7/1/000143/000143.html.

Easterling K (2012) *The Action Is the Form: Victor Hugo's TED Talk*. [Kindle version]. London, Moscow: Strelka Press.

Eclipse Foundation (2014) *Eclipse Community Survey 2014*. Available at: https://www.eclipse.org/org/community_survey/SurveySummary_2014-public.xls (accessed 5 August 2015).

Gabrys J (2014) Programming environments: Environmentality and citizen sensing in the smart city. *Environment and Planning D: Society and Space* 32(1): 30–48.

Ghosh RA, Glott R, Krieger B, et al. (2002) *Free/Libre and Open Source Software: Survey and Study*. Part 4: Survey of Developers. Maastricht: International Institute of Infonomics, University of Maastricht. Available at: www.flossproject.org/report/FLOSS_Final4.pdf (accessed 4 August 2015).

Graduiertenkolleg Topologie der Technik (2015) *Topological Manifesto*. Available at: https://www.tdt.tu-darmstadt.de/fileadmin/kolleg-tdt/Technospaces_2015/Manifest_FINAL_engl.pdf (accessed 30 June 2015).

Graham M (2011) Time machines and virtual portals: The spatialities of the digital divide. *Progress in Development Studies* 11(3): 211–227.

Graham SD (2005) Software-sorted geographies. *Progress in Human Geography* 29(5): 562–580.

Halpern O, LeCavalier J, Calvillo N, et al. (2013) Test-bed urbanism. *Public Culture* 25(270): 272–306.

Haraway D (1987) A manifesto for cyborgs: Science, technology, and socialist feminism in the 1980s. *Australian Feminist Studies* 2(4): 1–42.

Hayles K (1999) *How We Became Posthuman: Virtual Bodies in Cybernetics, Literature, and Informatics*. Chicago: University of Chicago Press.

Kitchin R (2015) Making sense of smart cities: Addressing present shortcomings. *Cambridge Journal of Regions, Economy and Society* 8(1): 131–136.

Kitchin R (2016) Code and the city: Reframing the conceptual terrain. In: Kitchin R and Perng S (eds) *Code and the City*. London: Routledge.

Kitchin R and Dodge M (2011) *Code/Space: Software and Everyday Life*. Cambridge, MA: MIT Press.

Kittler F (1995) *There Is No Software*. Available at: http://www.ctheory.net/articles.aspx?id=74 (accessed 9 May 2015).

Latham A (2002) Retheorizing the scale of globalization: Topologies, actor-networks, and cosmopolitanism. In: Herod A and Wright MW (eds) *Geographies of Power*. Oxford: Blackwell, pp. 115–144.

Latham A (2011) Topologies and the multiplicities of space-time. *Dialogues in Human Geography* 1(3): 312–315.

Latour B (2004) Why has critique run out of steam? From matters of fact to matters of concern. *Critical Inquiry* 30(2): 225–248.

Latour B (2005) *Reassembling the Social: An Introduction to Actor-Network-Theory*. Oxford, UK: Oxford University Press.

Latour B (2008) A cautius prometheus: A few steps toward a philosophy of design (with special attention to peter sloterdijk). *Keynote lecture for the Networks of Design meeting of the Design History Society at Falmouth*. Available at: http://smm.sagepub.com/ (accessed 30 October 2012).

Law J (2002) Objects and spaces. *Theory, Culture & Society* 19(5–6): 91–105.

Law J and Mol A (2001) Situating technoscience: An inquiry into spatialities. *Environment and Planning D: Society and Space* 19(5): 609–621.

Law J and Singleton V (2005) Object lessons. *Organization* 12(3): 331–355.

Lefebvre H (1991) *The Production of Space*. Oxford, UK: Blackwell.

Lessig L (2008) *Remix: Making Art and Commerce Thrive in the Hybrid Economy*. New York, NY: Penguin Press.

Lury C, Parisi L and Terranova T (2012) Introduction: The becoming topological of culture. *Theory, Culture & Society* 29(4–5): 3–35.

Mackenzie A (2016) Code-crowd: How software repositories express urban life. In: Kitchin R and Perng S (eds) *Code and the City*. London, UK: Routledge.

Manovich L (2013) *Software Takes Command: Extending the Language of New Media*. London, UK: Bloomsbury.

Marres N (2012) On some uses and abuses of topology in the social analysis of technology (or the problem with smart meters). *Theory, Culture & Society* 29(4–5): 288–310.

Martin L and Secor AJ (2014) Towards a post-mathematical topology. *Progress in Human Geography* 38(3): 420–438.

Mattern S (2014) Interfacing urban intelligence. *Places Journal*. Available at: https://placesjournal.org/article/interfacing-urban-intelligence/ (accessed 22 July 2015).

McFarlane C (2009) Translocal assemblages: Space, power and social movements. *Geoforum* 40(4): 561–567.

Merton RK (1968) *Social Theory and Social Structure*. New York, NY: Free Press.

Murdoch J (1998) The spaces of actor-network theory. *Geoforum* 29(4): 357–374.

Nafus D, Leach J and Krieger B (2006) *Free/Libre and Open Source Software: Survey and Study*. Gender: Integrated Report of Findings. Cambridge, UK: University of Cambridge. Available at: http://www.flosspols.org/deliverables/FLOSSPOLS-D16-Gender_Integrated_Report_of_Findings.pdf (accessed 4 August 2015).

Ong A and Collier SJ (2005) *Global Assemblages: Technology, Politics, and Ethics as Anthropological Problems*. Malden, MA: Blackwell.

Opitz S and Tellmann U (2012) Global territories: Zones of economic and legal dis/connectivity. *Distinktion: Scandinavian Journal of Social Theory* 13(3): 261–282.

Parikka J (2012) New materialism as media theory: Medianatures and dirty matter. *Communication and Critical/Cultural Studies* 9(1): 95–100.

Parisi L (2013) *Contagious Architecture: Computation, Aesthetics, and Space*. Cambridge, MA: MIT Press.

Pickering A (2002) Cybernetics and the mangle: Ashby, Beer and Pask. *Social Studies of Science* 32(3): 413–437.

Rogers R (2012) Mapping and the politics of web space. *Theory, Culture & Society* 29(4–5): 193–219.

Ruppert E (2012) The governmental topologies of database devices. *Theory, Culture & Society* 29(4–5): 116–136.

Serres M and Latour B (1995) *Conversations on Science, Culture, and Time*. Ann Arbor, MI: University of Michigan Press.

Sha XW (2012) Topology and morphogenesis. *Theory, Culture & Society* 29(4–5): 220–246.

Sheller M (2004) Mobile publics: Beyond the network perspective. *Environment and Planning D: Society and Space* 22(1): 39–52.

Shelton T, Zook M and Wiig A (2015) The 'actually existing smart city'. *Cambridge Journal of Regions, Economy and Society* 8(1): 13–25.

Shields R (2012) Cultural topology: The seven bridges of Konigsburg [sic], 1736. *Theory, Culture & Society* 29(4–5): 43–57.

Solomon R (2013) Last in, first out: Network archeology as/of the stack. *Amodern* 2. Available at: http://amodern.net/article/last-in-first-out/ (accessed 27 August 2015).

Star SL (1999) The ethnography of infrastructure. *American Behavioral Scientist* 43(3): 377–391.

Terranova T (2014) Red stack attack! Algorithms, capital and the automation of the common. In: Mackay R and Avanessian A (eds) *#Accelerate#*. Falmouth, UK; Berlin, DE: Urbanomic Media; Merve.

Thrift N and French S (2002) The automatic production of space. *Transactions of the Institute of British Geographers* 27(3): 309–335.

van Dijk JAGM (2006) Digital divide research, achievements and shortcomings. *Poetics* 34(4–5): 221–235.

White JM (2016) Moving applications: A multilayered approach to mobile computing. In: Kitchin R and Perng S (eds) *Code and the City*. London, UK: Routledge.

Wilson MW (2012) Location-based services, conspicuous mobility, and the location-aware future. *Geoforum* 43(6): 1266–1275.

Wyatt S and Balmer B (2007) Home on the range: what and where is the middle in science and technology studies? *Science, Technology & Human Values* 32(6): 619–626.

Zimmermann H (1980) OSI reference model: The ISO model of architecture for open systems interconnection. *IEEE Transactions on Communications* 28(4): 425–432.

Zittrain J (2008) *The Future of the Internet—and How to Stop It*. New Haven, CT: Yale University Press.

Zook MA and Graham M (2007) Mapping DigiPlace: Geocoded Internet data and the representation of place. *Environment and Planning B: Planning and Design* 34(3): 466–482.

Appendix

Documents used in reconstructing the history of Git

Barr J and Torvalds L (2005) *BitKeeper and Linux: The End of the Road?* Available at: http://archive09.linux.com/feature/44147 (accessed 17 August 2015).

Chacon S and Straub B (2014) *Pro Git: Everything You Need to Know about Git*. New York, NY: Apress.

Cloer J and Torvalds L (2015) *10 Years of Git: An Interview with Git Creator Linus Torvalds*. Available at: http://www.linux.com/news/featured-blogs/185-jennifer-cloer/821541-10-years-of-git-an-interview-with-git-creator-linus-torvalds (accessed 17 August 2015).

Git (2015) *Git(1) Manual Page*. Available at: https://www.kernel.org/pub/software/scm/git/docs/ (accessed 17 August 2015).

Hamano JC (2006) Git: A stupid content tracker. In: *Proceedings of the Ottawa Linux Symposium 2006*. Available at: http://download.hforge.org/doc/git-resources/200607-ols.pdf (accessed 17 August 2015).

Hamano JC (2008) Git Chronicles: How much of this did linus really write? In: *Proceedings of the GitTogether conference 2008*, Mountain View. Available at: https://docs.google.com/file/d/0Bw3FApcOlPDhMFR3UldGSHFGcjQ/view (accessed 17 August 2015).

Jorgensen G and Torvalds L (2012) *Linus Torvalds Goes off on Linux and Git*. Available at: http://typical-programmer.com/linus-torvalds-goes-off-on-linux-and-git/ (accessed 17 August 2015).

Knorr E and Preston-Werner T (2012) *GitHub CEO: We're Helping Software Eat the World*. Available at: http://www.infoworld.com/article/2615989/application-development/github-ceo–we-re-helping-software-eat-the-world.html (accessed 17 August 2015).

McMillan R (2005) *After Controversy, Torvalds Begins Work on 'Git'*. Available at: http://www.pcworld.idg.com.au/article/129776/after_controversy_torvalds_begins_work_git_/ (accessed 17 August 2015).

McMillan R (2012a) *Lord of the Files: How GitHub Tamed Free Software (And More)*. Available at: http://www.wired.com/2012/02/github-2/ (accessed 17 August 2015).

McMillan R (2012b) *The Legacy of Linus Torvalds: Linux, Git, and One Giant Flamethrower*. Available at:

http://www.wired.com/2012/11/linus-torvalds-isoc/ (accessed 17 August 2015).

Neumann A (2015) *Vor 10 Jahren: Linus Torvalds baut Git*. Available at: http://www.heise.de/developer/meldung/Vor-10-Jahren-Linus-Torvalds-baut-Git-2596654.html (accessed 17 August 2015).

Orsini L and Preston-Werner T (2013) *GitHub's Tom Preston-Werner: How We Went Mainstream*. Available at: http://readwrite.com/2013/11/18/github-tom-preston-warner (accessed 17 August 2015).

Torvalds L (2005a) *'First ever real kernel git merge!' [E-Mail to Git Mailing List]*. Available at: http://marc.info/?l=git&m=111377572329534 (accessed 17 August 2015).

Torvalds L (2005b) *'Kernel SCM saga.' [E-Mail to Linux-Kernel Mailing List]*. Available at: http://marc.info/?l=linux-kernel&m=111280216717070 (accessed 17 August 2015).

Torvalds L (2005c) *'Re: Kernel SCM saga.' [E-Mail to Linux Kernel Mailing List]*. Available at: http://marc.info/?l=linux-kernel&m=111293537202443 (accessed 17 August 2015).

Torvalds L (2005d) *'Re: Kernel SCM saga.' [E-Mail to Linux-Kernel Mailing List]*. Available at: http://marc.info/?l=linux-kernel&m=111288700902396 (accessed 17 August 2015).

Torvalds L (2005e) *'Re: more git updates.' [E-mail to Linux Kernel Mailing List]*. Available at: http://marc.info/?l=linux-kernel&m=111314792424707 (accessed 17 August 2015).

Torvalds L (2006a) *'Re: git and bzr' [E-mail to Git Mailing List]*. Available at: http://marc.info/?l=git&m=116473016012824 (accessed 17 August 2015).

Torvalds L (2006b) *'Re: VCS comparison table' [E-mail to Git Mailing List]*. Available at: http://marc.info/?l=git&m=116128307511686 (accessed 17 August 2015).

Torvalds L (2006c) *'Re: VCS comparison table' [E-Mail to Git and Bazaar Mailing Lists]*. Available at: http://marc.info/?l=git&m=116129092117475&w=2 (accessed 17 August 2015).

Torvalds L (2006d) *'Re: [ANNOUNCE] Git wiki' [E-Mail to Git Mailing List]*. Available at: http://marc.info/?l=git&m=114685143200012 (accessed 6 August 2015).

Torvalds L (2007a) *'Re: Trivia: When did git self-host?' [E-mail to Git Mailing List]*. Available at: http://marc.info/?l=git&m=117254154130732 (accessed 17 August 2015).

Torvalds L (2007b) *Linus Torvalds on Git [Google Tech Talk]*. Available at: https://www.youtube.com/watch?v=4XpnKHJAok8 (accessed 17 August 2015).

Zemlin J (2012) *The Greatness of Git*. Available at: http://www.linuxfoundation.org/news-media/blogs/browse/2012/02/greatness-git (accessed 17 August 2015).

This article is a part of special theme on *Spatial Big Data*. To see a full list of all articles in this special theme, please click here: http://bds.sagepub.com/content/spatial-big-data.