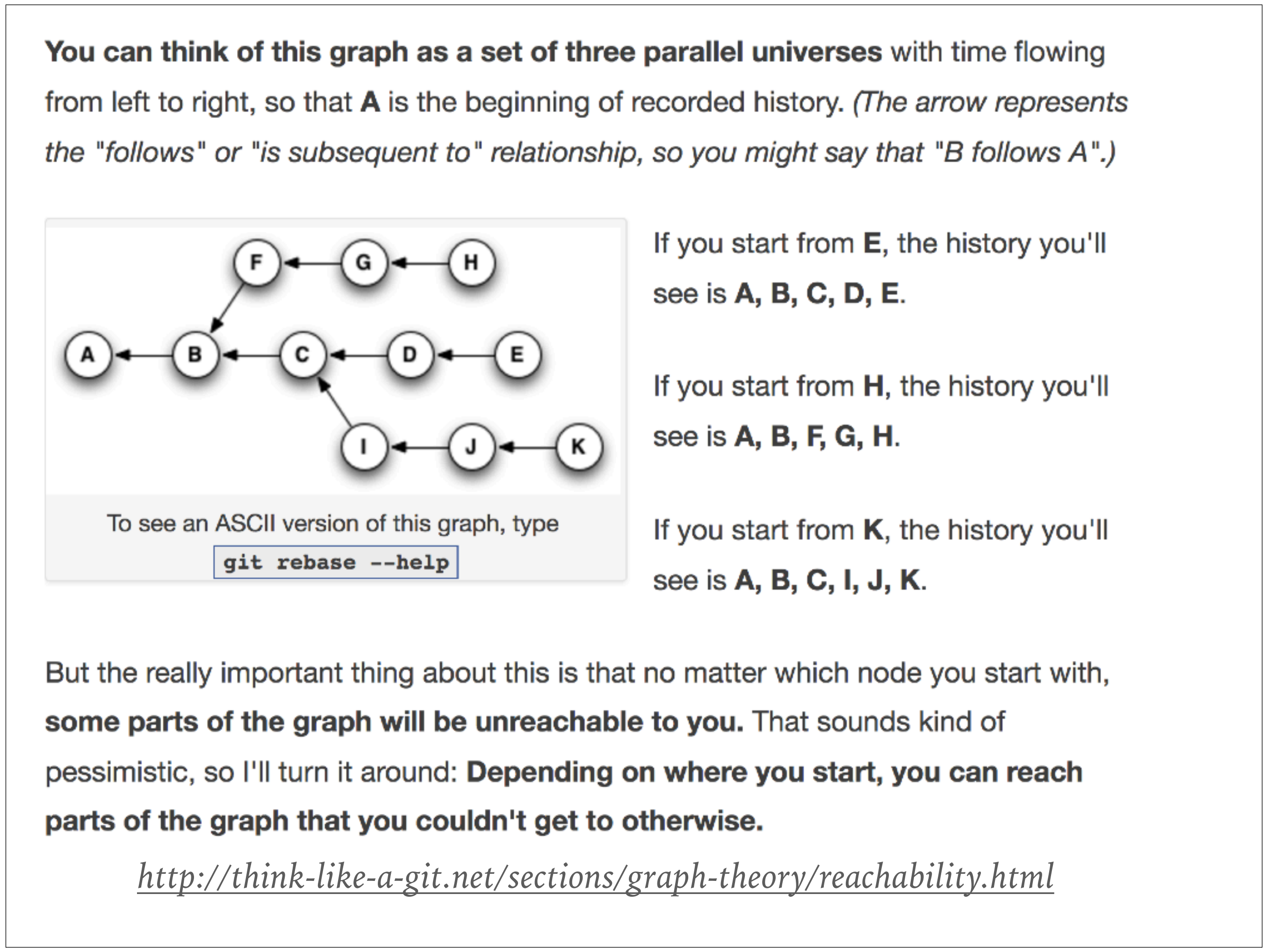
GIT: FROM SCRATCH

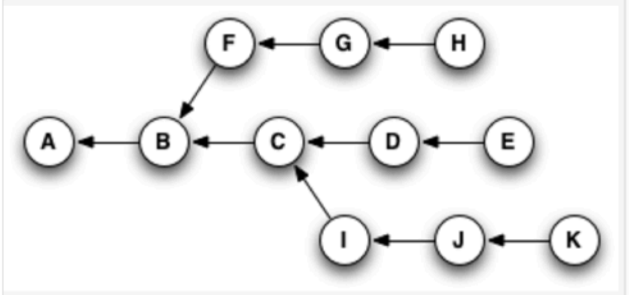
for beginners and mavens alike

START WITH A MEME

There’s an old joke format about git: git makes much more sense when you understand X.



You can think of this graph as a set of three parallel universes with time flowing from left to right, so that **A** is the beginning of recorded history. *(The arrow represents the "follows" or "is subsequent to" relationship, so you might say that "B follows A".)*



To see an ASCII version of this graph, type `git rebase --help`

If you start from **E**, the history you'll see is **A, B, C, D, E**.

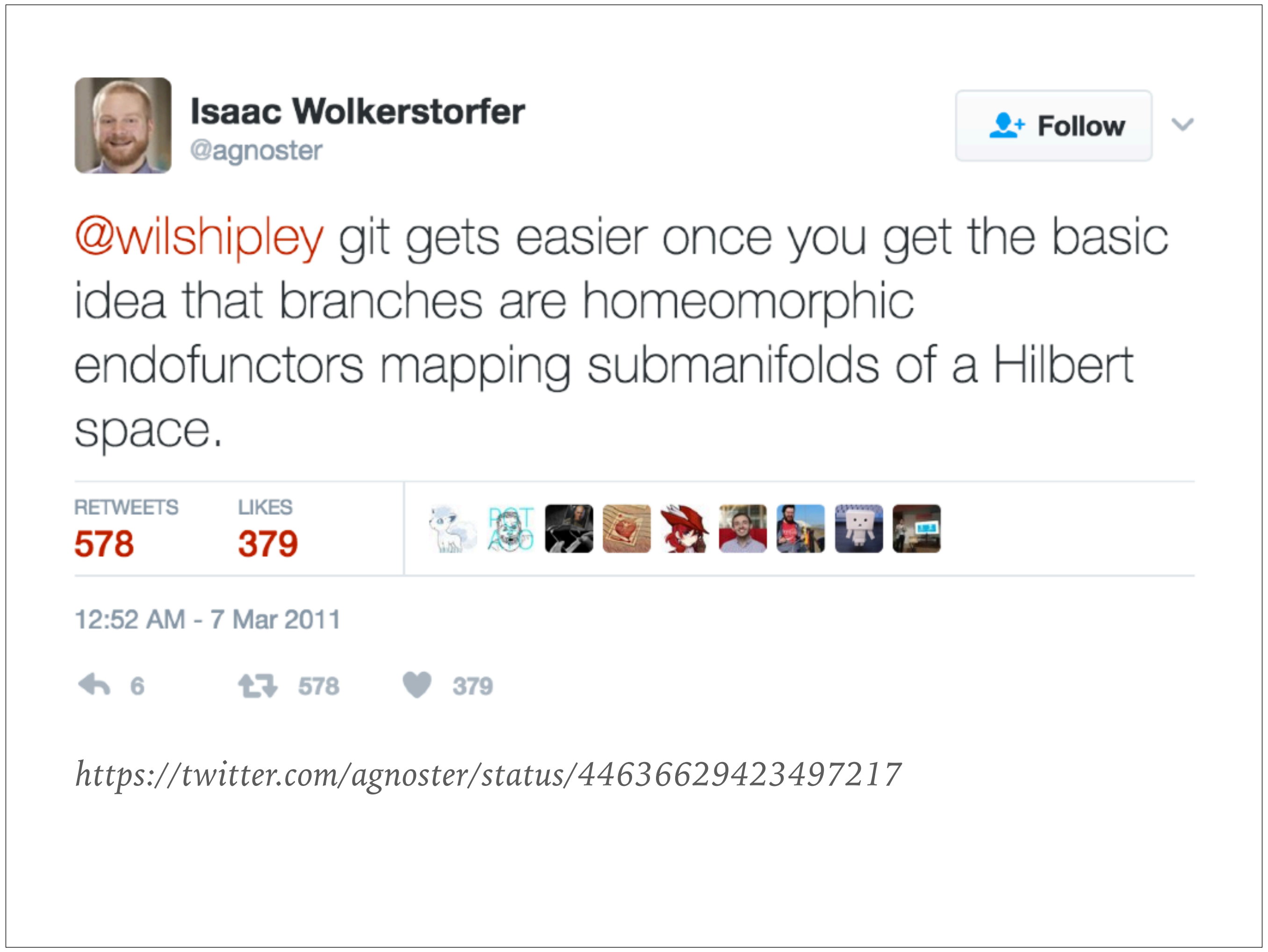
If you start from **H**, the history you'll see is **A, B, F, G, H**.

If you start from **K**, the history you'll see is **A, B, C, I, J, K**.

But the really important thing about this is that no matter which node you start with, **some parts of the graph will be unreachable to you.** That sounds kind of pessimistic, so I'll turn it around: **Depending on where you start, you can reach parts of the graph that you couldn't get to otherwise.**

http://think-like-a-git.net/sections/graph-theory/reachability.html

This starts pretty sincerely from the quite true assertion that it can really help to understand that, underlying git, there is a graph data structure, and that a series of commits is a sequence nodes pointing to their parents.



Isaac Wolkerstorfer
@agnoster

Follow

@wilshipley git gets easier once you get the basic idea that branches are homeomorphic endofunctors mapping submanifolds of a Hilbert space.

RETWEETS 578 LIKES 379

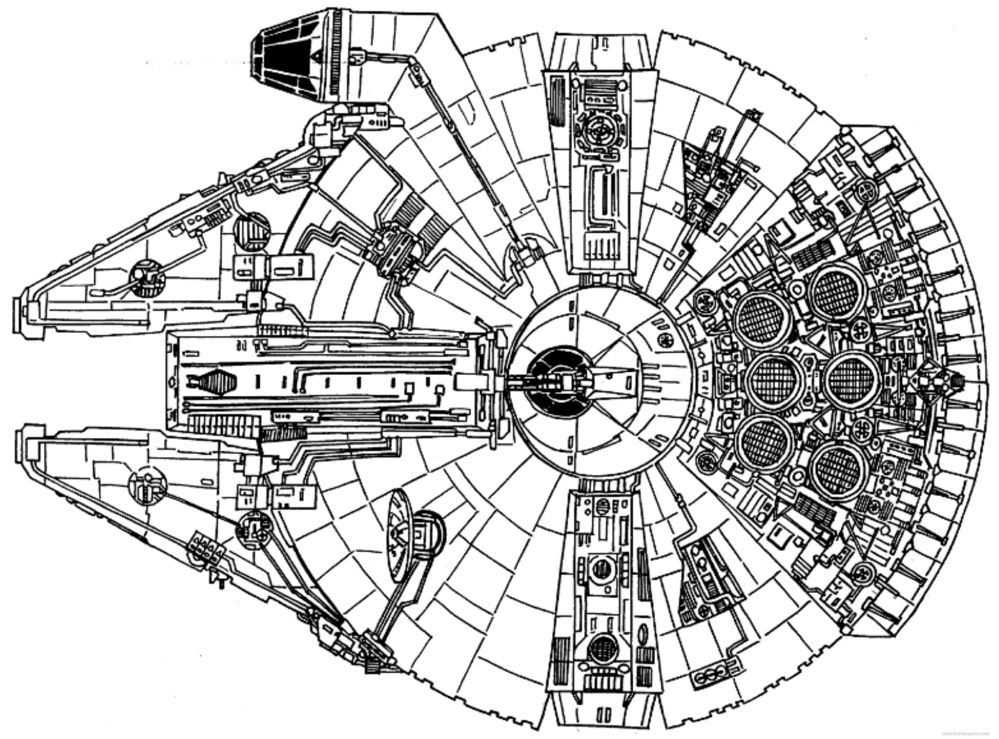
12:52 AM - 7 Mar 2011

6 578 379

https://twitter.com/agnoster/status/44636629423497217

But this is the original, classic joke. As git started becoming more popular, smart people tried to explain it to other people and git was complicated enough that they had to compare it with other complicated things they knew, in order to explain it.

The truth behind this joke is that you will never have proficiency with git simply by learning and memorizing the commands. You need to understand what they are doing to the underlying data structures, and it helps to understand why those data structures are the way they are.



"You came in that thing? You're braver than I thought."

—Leia Organa

Git at its core really is these graph manipulation commands, and that is what makes it so useful and flexible. But around this core is an accretion of features and improvements that can be somewhat ad hoc, and can be utterly incomprehensible if you don’t know how things came to be the way they are.

My goal today is not to make you an expert in git in one go. My goal is to give you an understanding of the history and a deep understanding of the core concepts, so that you know where to look, and so that when you hit a situation you don’t know how to fathom, you may be equipped to investigate in a way that both solves your problem and makes you better at git.

THE PLAN FOR TODAY

- a brief historical background
- a hands-on walkthrough of the underlying data structures (**plumbing**) and the command-line interfaces (**porcelain**)
  - (plumbing/porcelain is standard git terminology)

- The question I'd like you to keep in mind:

HOW WOULD YOU WRITE A VERSION CONTROL SYSTEM?

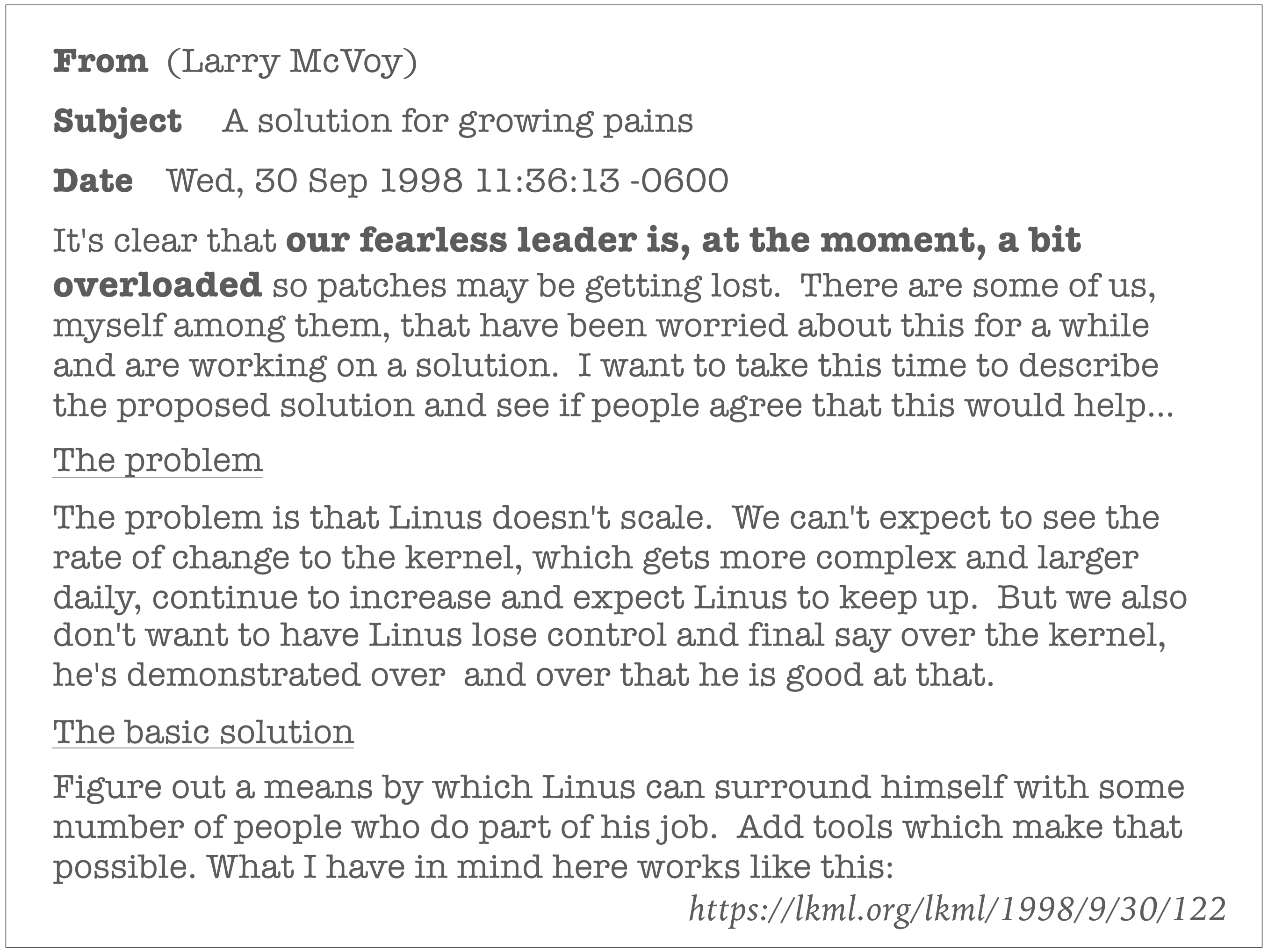
you could, today, with the data structures you know, if you really had to.

HOW WOULD YOU WRITE A VCS?

- The problem space:
  - Files: *several*
  - Revisions: *ongoing*
  - History: *important*
  - Code: *inherently buggy, needs lots of little changes*
  - Branching??? *what even is*

GIT: ORIGINS, VOL. 1: BITKEEPER

- Linux kernel development, 1992-1998: code changes are passed around as emailed **patches**
  - (a patch is the output of a line-by-line `diff` on a file, plus some context so it can be automatically applied)
- Linus Torvalds accretes a small group of trusted code-reviewers who review patches, provisionally approve them, and then send them to Linus for final approval and adding to the kernel
- VCSes exist, but **Linus hates them** and doesn't use one. `Subversion` repos of kernel source are purely a distribution convenience.
- Community collaboration happens on LKML


From (Larry McVoy)

Subject A solution for growing pains

Date Wed, 30 Sep 1998 11:36:13 -0600

It's clear that **our fearless leader is, at the moment, a bit overloaded** so patches may be getting lost. There are some of us, myself among them, that have been worried about this for a while and are working on a solution. I want to take this time to describe the proposed solution and see if people agree that this would help...

The problem

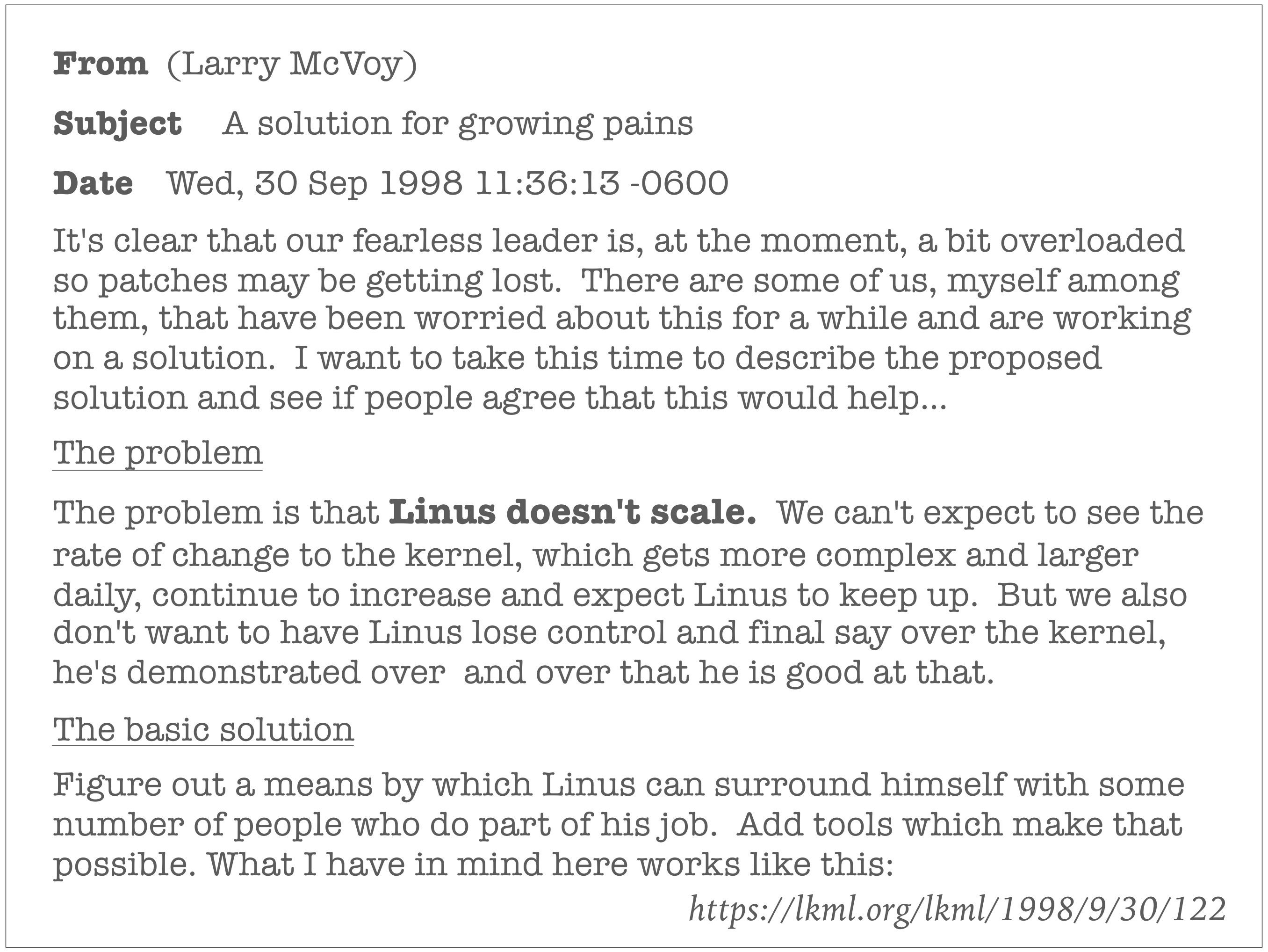
The problem is that Linus doesn't scale. We can't expect to see the rate of change to the kernel, which gets more complex and larger daily, continue to increase and expect Linus to keep up. But we also don't want to have Linus lose control and final say over the kernel, he's demonstrated over and over that he is good at that.

The basic solution

Figure out a means by which Linus can surround himself with some number of people who do part of his job. Add tools which make that possible. What I have in mind here works like this:

https://lkml.org/lkml/1998/9/30/122

Managing large software collaborations is hard


From (Larry McVoy)

Subject A solution for growing pains

Date Wed, 30 Sep 1998 11:36:13 -0600

It's clear that our fearless leader is, at the moment, a bit overloaded so patches may be getting lost. There are some of us, myself among them, that have been worried about this for a while and are working on a solution. I want to take this time to describe the proposed solution and see if people agree that this would help...

The problem

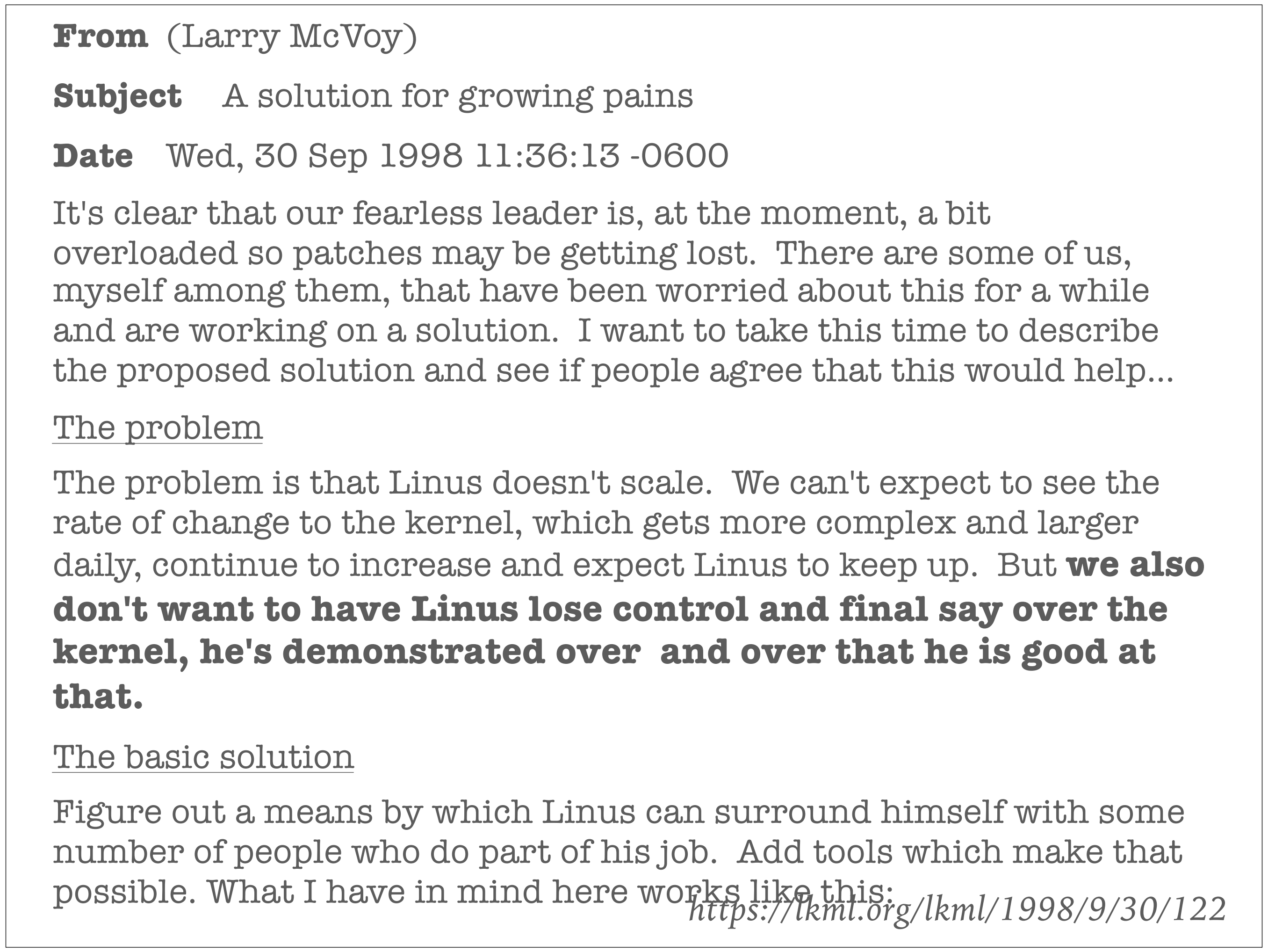
The problem is that **Linus doesn't scale.** We can't expect to see the rate of change to the kernel, which gets more complex and larger daily, continue to increase and expect Linus to keep up. But we also don't want to have Linus lose control and final say over the kernel, he's demonstrated over and over that he is good at that.

The basic solution

Figure out a means by which Linus can surround himself with some number of people who do part of his job. Add tools which make that possible. What I have in mind here works like this:

https://lkml.org/lkml/1998/9/30/122

Putting all the work in a single point of failure is risky



From (Larry McVoy)

Subject A solution for growing pains

Date Wed, 30 Sep 1998 11:36:13 -0600

It's clear that our fearless leader is, at the moment, a bit overloaded so patches may be getting lost. There are some of us, myself among them, that have been worried about this for a while and are working on a solution. I want to take this time to describe the proposed solution and see if people agree that this would help...

The problem

The problem is that Linus doesn't scale. We can't expect to see the rate of change to the kernel, which gets more complex and larger daily, continue to increase and expect Linus to keep up. But **we also don't want to have Linus lose control and final say over the kernel, he's demonstrated over and over that he is good at that.**

The basic solution

Figure out a means by which Linus can surround himself with some number of people who do part of his job. Add tools which make that possible. What I have in mind here works like this:

https://lkml.org/lkml/1998/9/30/122

It’s usually necessary to let multiple people review changes before officially making them part of the project

Suppose **J Random Hacker makes a change to the kernel and posts a patch.** It's a questionable patch so Linus just ignores it. But suppose somebody else a little less random wants to try it out. So they grab it, apply it to their tree, tweak it a little and it works. So they post an update patch that includes both their tweaks plus their comments. Now one of the first tier folks like Dave or Alan grab it and stuff it into their tree. At this point perhaps the patch works fine as is. So they add their comments to the patch and send it to Linus.

What Linus gets is a specific patch with a patch history that shows that this patch has been seen to work by several people other than the developer, including some people that Linus trusts to have good judgement. It still doesn't mean th patch gets applied, it just means that Linus has more information in a self contained package.

If he decides to apply it, the patch goes in and the comment history also gets squirreled away as well.

Details

The mechanism which allows all this to happen is a distributed source management system...

https://lkml.org/lkml/1998/9/30/122

Suppose J Random Hacker makes a change to the kernel and posts a patch. It's a questionable patch so Linus just ignores it. But suppose **somebody else** a little less random wants to try it out. So they **grab it, apply it to their tree, tweak it a little and it works. So they post an update patch that includes both their tweaks plus their comments.** Now one of the first tier folks like Dave or Alan grab it and stuff it into their tree. At this point perhaps the patch works fine as is. So they add their comments to the patch and send it to Linus.

What Linus gets is a specific patch with a patch history that shows that this patch has been seen to work by several people other than the developer, including some people that Linus trusts to have good judgement. It still doesn't mean th patch gets applied, it just means that Linus has more information in a self contained package.

If he decides to apply it, the patch goes in and the comment history also gets squirreled away as well.

Details

The mechanism which allows all this to happen is a distributed source management system...

https://lkml.org/lkml/1998/9/30/122

Suppose J Random Hacker makes a change to the kernel and posts a patch. It's a questionable patch so Linus just ignores it. But suppose somebody else a little less random wants to try it out. So they grab it, apply it to their tree, tweak it a little and it works. So they post an update patch that includes both their tweaks plus their comments. Now one of the first tier folks like Dave or Alan grab it and stuff it into their tree. At this point perhaps the patch works fine as is. So they add their comments to the patch and send it to Linus.

What Linus gets is **a specific patch with a patch history that shows that this patch has been seen to work by several people other than the developer, including some people that Linus trusts to have good judgement.** It still doesn't mean the patch gets applied, it just means that Linus has more information in a self contained package.

If he decides to apply it, the patch goes in and the comment history also gets squirreled away as well.

Details

The mechanism which allows all this to happen is a distributed source management system...

https://lkml.org/lkml/1998/9/30/122

Suppose J Random Hacker makes a change to the kernel and posts a patch. It's a questionable patch so Linus just ignores it. But suppose somebody else a little less random wants to try it out. So they grab it, apply it to their tree, tweak it a little and it works. So they post an update patch that includes both their tweaks plus their comments. Now one of the first tier folks like Dave or Alan grab it and stuff it into their tree. At this point perhaps the patch works fine as is. So they add their comments to the patch and send it to Linus.

What Linus gets is a specific patch with a patch history that shows that this patch has been seen to work by several people other than the developer, including some people that Linus trusts to have good judgement. It still doesn't mean the patch gets applied, it just means that Linus has more information in a self contained package.

If he decides to apply it, the patch goes in and **the comment history also gets squirreled away as well.**

Details

The mechanism which allows all this to happen is a distributed source management system...

https://lkml.org/lkml/1998/9/30/122

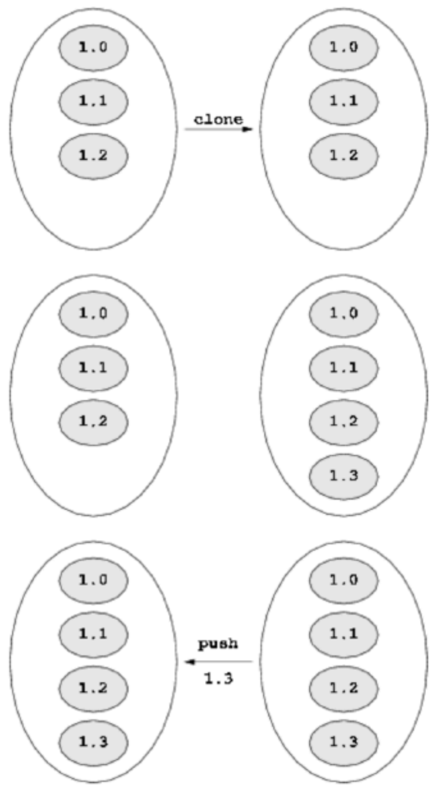
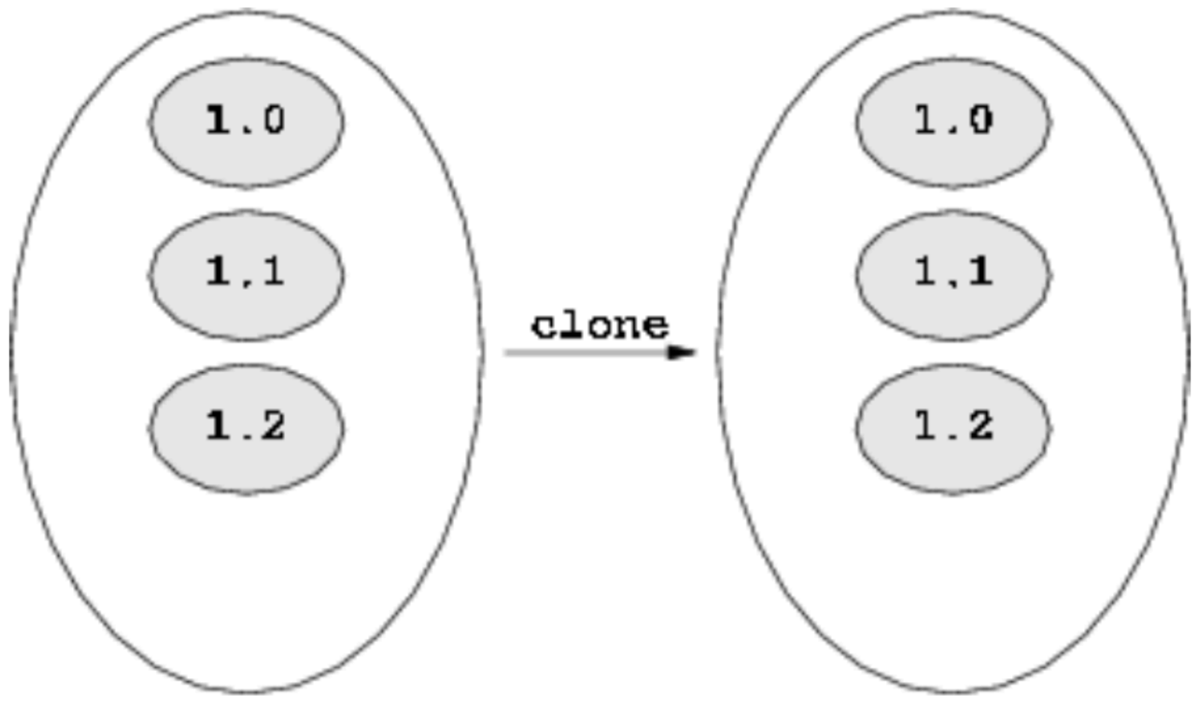


Figure 4: Example of a push: Initial clone, commit a change, push it back.

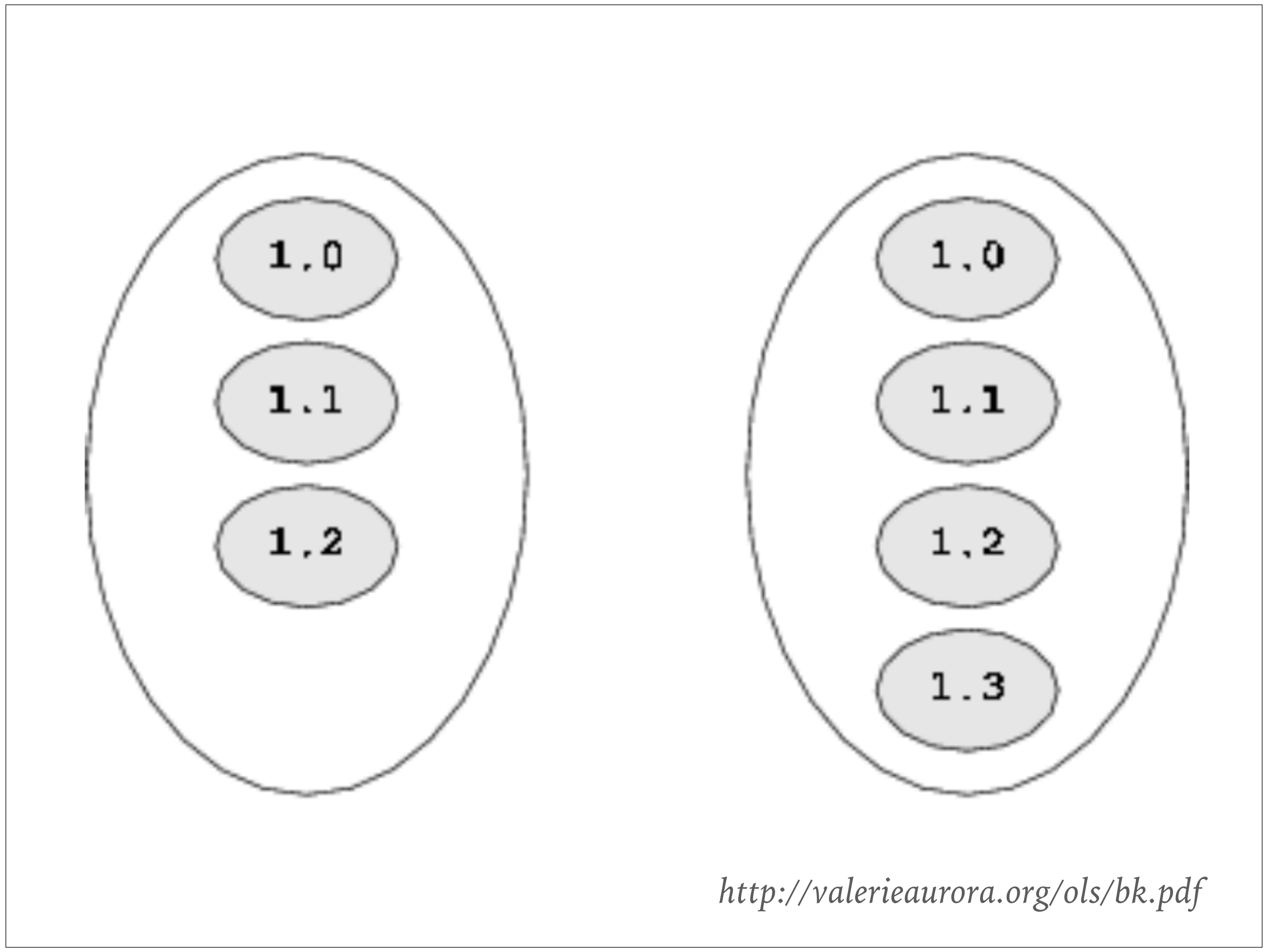
http://valerieaurora.org/ols/bk.pdf

Bitkeeper had cloning, pushing and pulling:

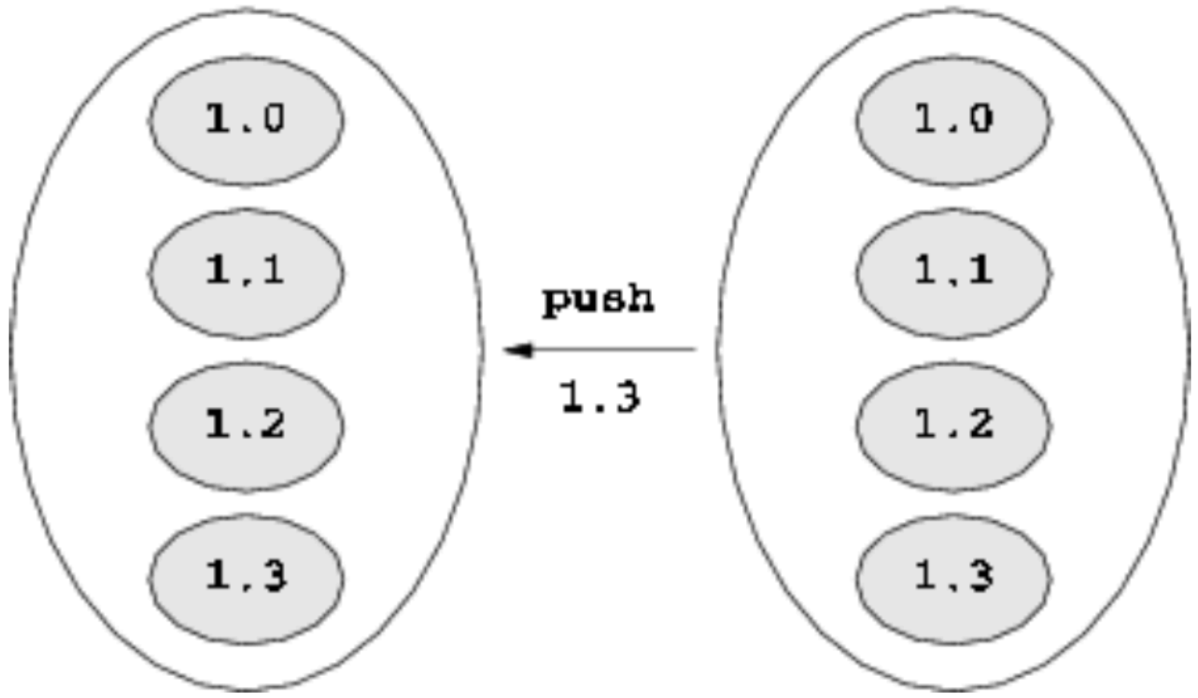


http://valerieaurora.org/ols/bk.pdf

you’d start by cloning an upstream repo,



you’d make changes in your local repo



http://valerieaurora.org/ols/bk.pdf

and then you'd push them back upstream and they'd be in the repo

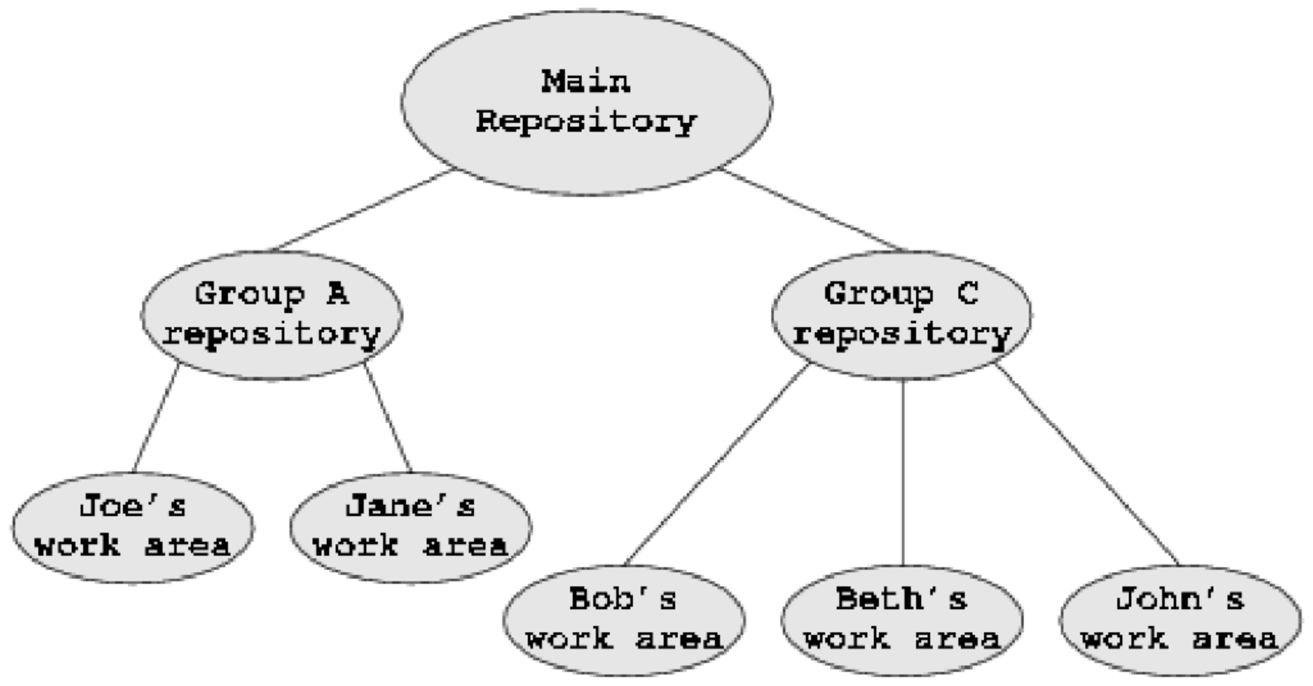


Figure 3: Example BitKeeper repository structure.

http://valerieaurora.org/ols/bk.pdf

And it had this thing that we have in Git, where you can have branches, and branches of branches

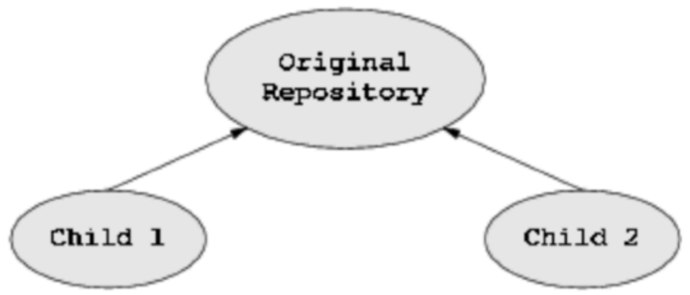


Figure 1: Parent pointers after cloning.

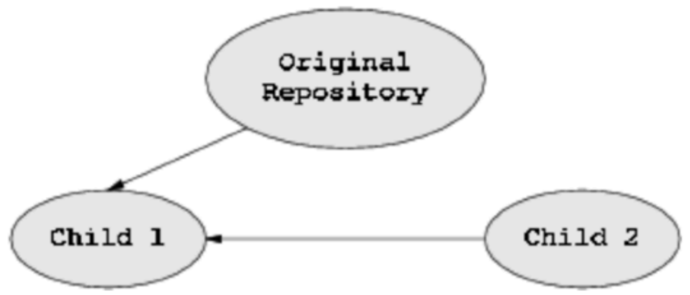


Figure 2: Parent pointers after changing with “`bk parent`”.

http://valerieaurora.org/ols/bk.pdf

And it even had some of the less frequently-used features, like being able to arbitrarily change what one checkout’s upstream ‘source of truth’ is

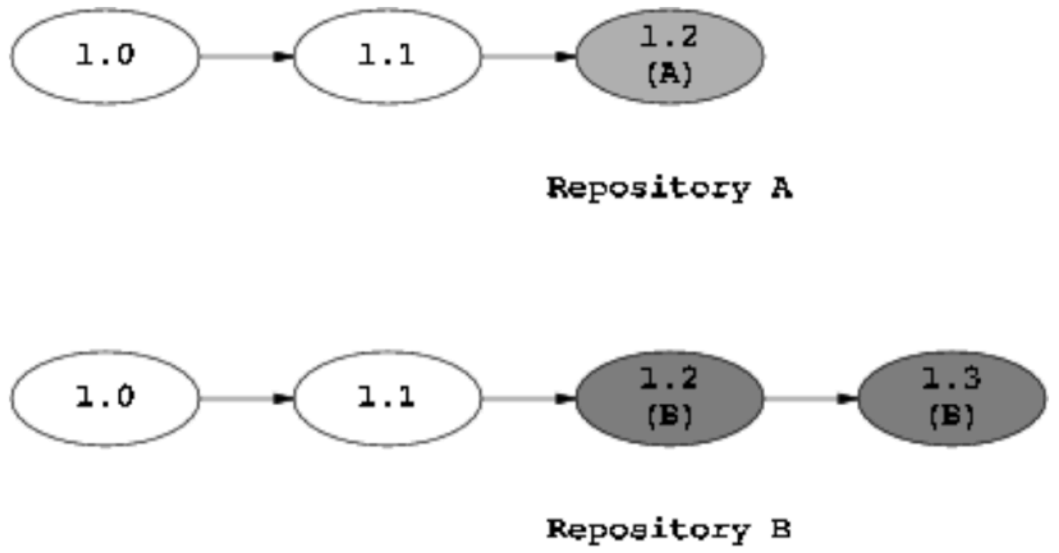


Figure 5: Repositories before merge, shaded changesets were added since clone.

And it had this notion of merging atomic commits in sequence: so Alice checked out repo A, and made a commit to create a new version locally; and Billie checked out repo B, did two commits. But now it's time to merge. How do you do that?

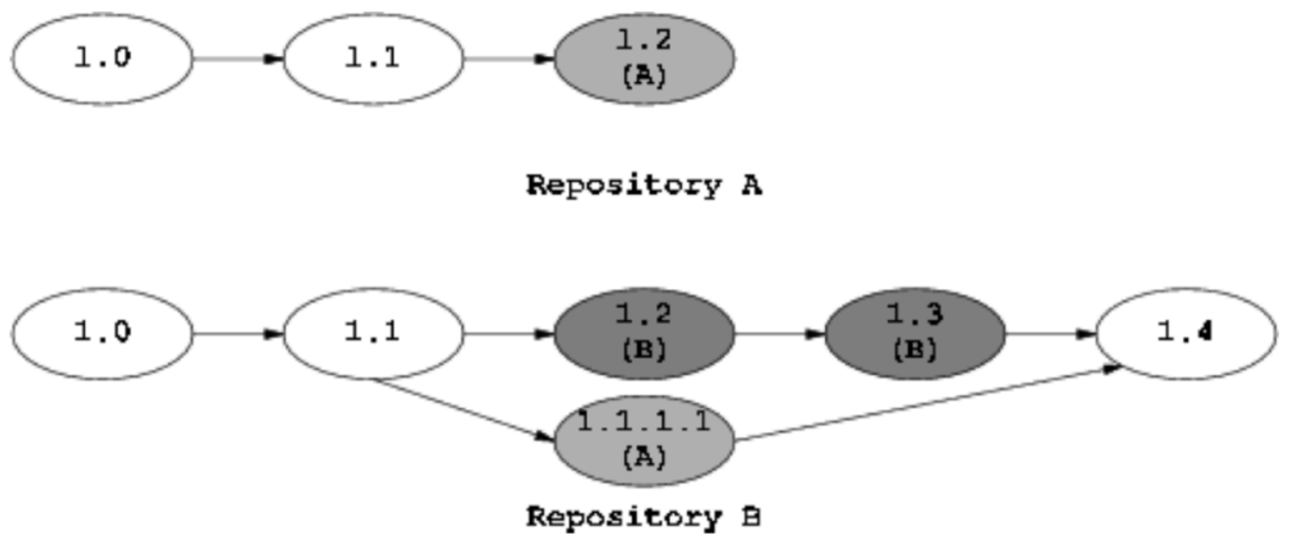


Figure 6: After a pull of A’s changes to B, with A’s changes merged.

Torvalds: I really never wanted to do source control management at all and felt that it was just about the least interesting thing in the computing world (with the possible exception of databases ;^), and I hated all SCM's with a passion… BitKeeper came along and really changed the way I viewed source control… **having a local copy of the repository and distributed merging was a big deal.** The big thing about distributed source control is that it makes one of the main issues with SCM's go away - the politics around "who can make changes." BK showed that you can avoid that by just giving everybody their own source repository. …[T]he biggest downside was the fact that since it wasn't open source, there was a lot of people who didn't want to use it. So while we ended up having several core maintainers use BK - it was free to use for open source projects - it never got ubiquitous. So it helped kernel development, but there were still pain points.

That then came to a head when [Andrew Tridgell] started reverse-engineering the (fairly simply) BK protocol, which was against the usage rules for BK.

https://www.linux.com/blog/10-years-git-interview-git-creator-linus-torvalds

Torvalds: I really never wanted to do source control management at all and felt that it was just about the least interesting thing in the computing world (with the possible exception of databases ;^), and I hated all SCM's with a passion… BitKeeper came along and really changed the way I viewed source control… having a local copy of the repository and distributed merging was a big deal. The big thing about distributed source control is that it makes one of the main issues with SCM's go away - **the politics around "who can make changes." BK showed that you can avoid that by just giving everybody their own source repository.** …[T]he biggest downside was the fact that since it wasn't open source, there was a lot of people who didn't want to use it. So while we ended up having several core maintainers use BK - it was free to use for open source projects - it never got ubiquitous. So it helped kernel development, but there were still pain points.

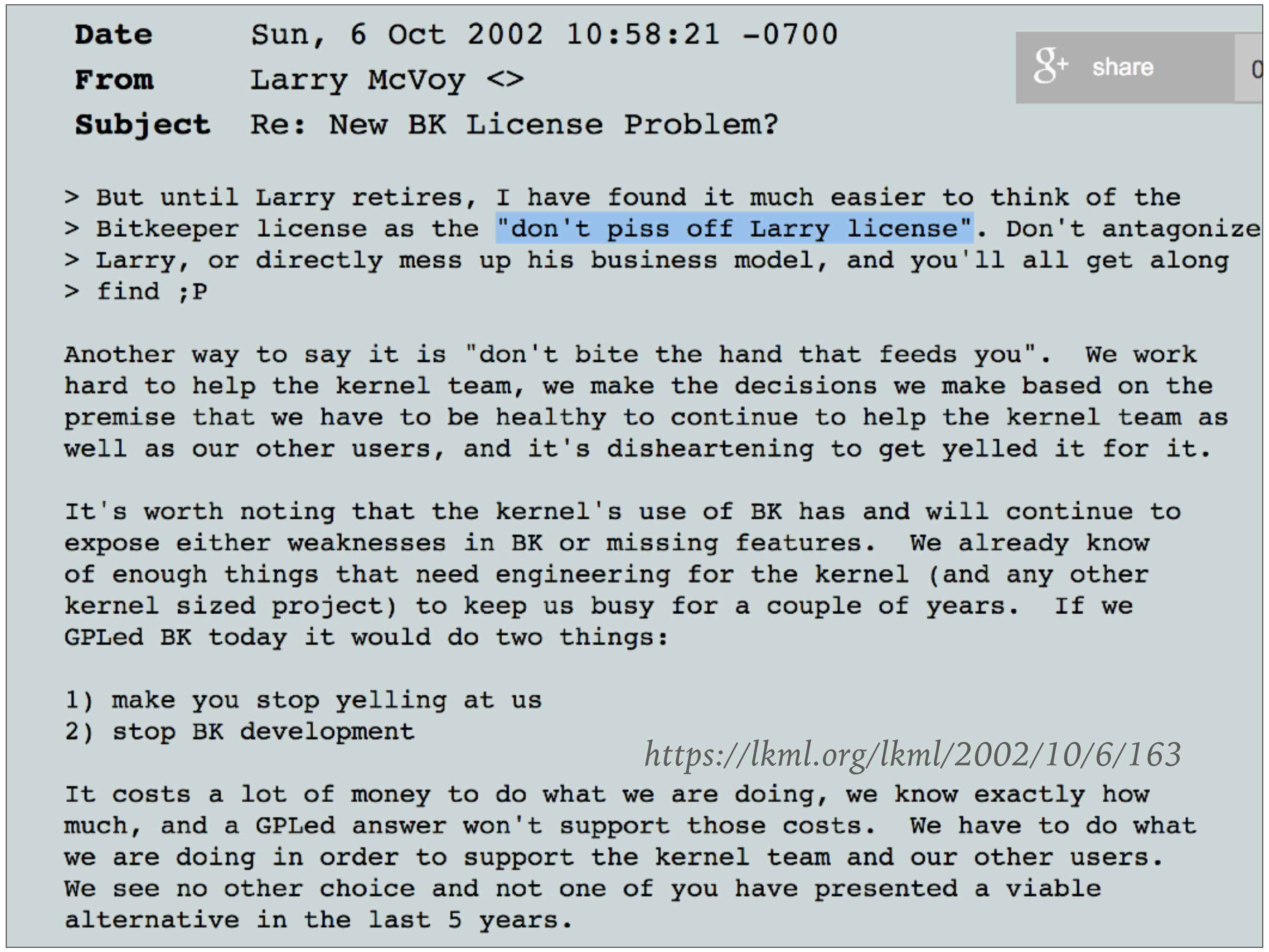
That then came to a head when [Andrew Tridgell] started reverse-engineering the (fairly simply) BK protocol, which was against the usage rules for BK.

https://www.linux.com/blog/10-years-git-interview-git-creator-linus-torvalds

Torvalds: I really never wanted to do source control management at all and felt that it was just about the least interesting thing in the computing world (with the possible exception of databases ;^), and I hated all SCM's with a passion... BitKeeper came along and really changed the way I viewed source control... having a local copy of the repository and distributed merging was a big deal. The big thing about distributed source control is that it makes one of the main issues with SCM's go away - the politics around "who can make changes." BK showed that you can avoid that by just giving everybody their own source repository. ...[T]he biggest downside was the fact that since it wasn't open source, there was a lot of people who didn't want to use it. So while we ended up having several core maintainers use BK - it was free to use for open source projects - it never got ubiquitous. So it helped kernel development, but there were still pain points.

That then came to a head when [Andrew Tridgell] started reverse-engineering the (fairly simply) BK protocol, which was against the usage rules for BK.

https://www.linux.com/blog/10-years-git-interview-git-creator-linus-torvalds

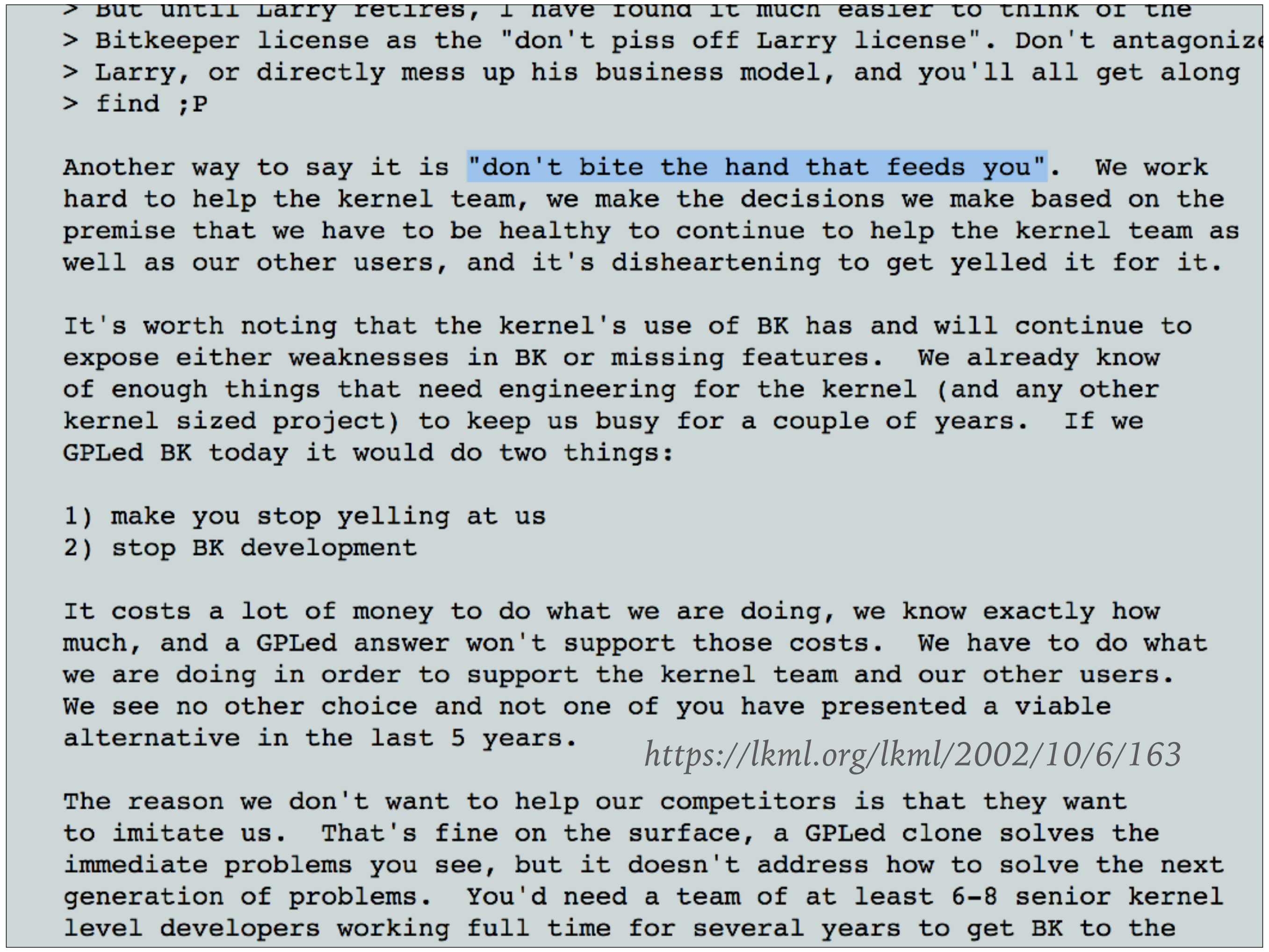


```
Date      Sun, 6 Oct 2002 10:58:21 -0700
From      Larry McVoy <>
Subject   Re: New BK License Problem?

> But until Larry retires, I have found it much easier to think of the
> Bitkeeper license as the "don't piss off Larry license". Don't antagonize
> Larry, or directly mess up his business model, and you'll all get along
> find ;P

Another way to say it is "don't bite the hand that feeds you".  We work
hard to help the kernel team, we make the decisions we make based on the
premise that we have to be healthy to continue to help the kernel team as
well as our other users, and it's disheartening to get yelled it for it.

It's worth noting that the kernel's use of BK has and will continue to
expose either weaknesses in BK or missing features.  We already know
of enough things that need engineering for the kernel (and any other
kernel sized project) to keep us busy for a couple of years.  If we
GPLed BK today it would do two things:

1) make you stop yelling at us
2) stop BK development

It costs a lot of money to do what we are doing, we know exactly how
much, and a GPLed answer won't support those costs.  We have to do what
we are doing in order to support the kernel team and our other users.
We see no other choice and not one of you have presented a viable
alternative in the last 5 years.
```

https://lkml.org/lkml/2002/10/6/163

Larry had these rules because he didn't want this incredibly productive and energized community to drink his milkshake.

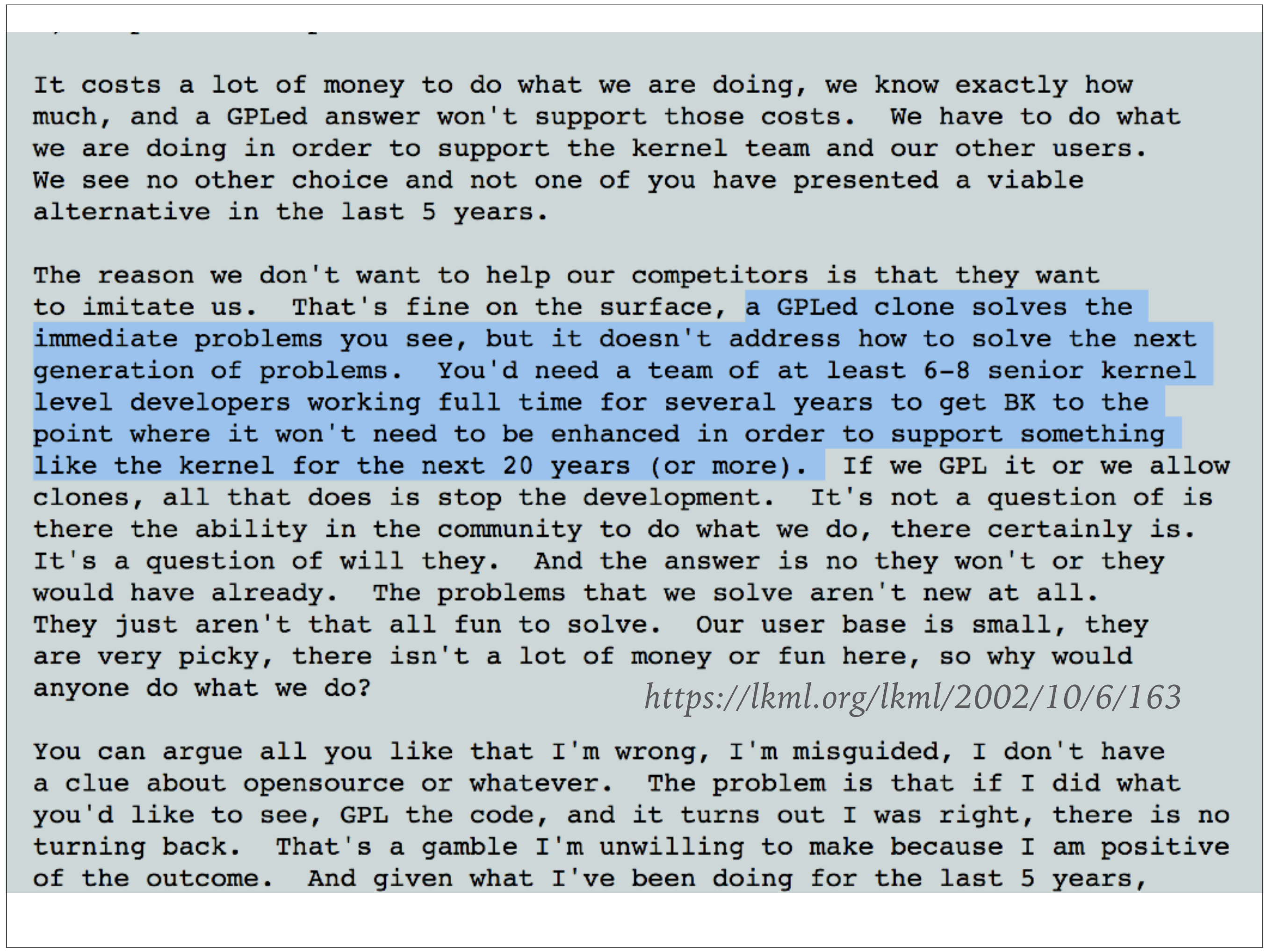
Now there's a fun, dramatic story here about how the BK thing fell apart, and one of the lessons is that people will think of all sorts of reasons why *their* code is special, and no one will ever come up with a community-created version of their thing that is as good as what a proprietary commercial thing is



```
> But until Larry retires, I have found it much easier to think of the
> Bitkeeper license as the "don't piss off Larry license". Don't antagonize
> Larry, or directly mess up his business model, and you'll all get along
> find ;P

Another way to say it is "don't bite the hand that feeds you".  We work
hard to help the kernel team, we make the decisions we make based on the
premise that we have to be healthy to continue to help the kernel team as
well as our other users, and it's disheartening to get yelled it for it.

It's worth noting that the kernel's use of BK has and will continue to
expose either weaknesses in BK or missing features.  We already know
of enough things that need engineering for the kernel (and any other
kernel sized project) to keep us busy for a couple of years.  If we
GPLed BK today it would do two things:

1) make you stop yelling at us
2) stop BK development

It costs a lot of money to do what we are doing, we know exactly how
much, and a GPLed answer won't support those costs.  We have to do what
we are doing in order to support the kernel team and our other users.
We see no other choice and not one of you have presented a viable
alternative in the last 5 years.

The reason we don't want to help our competitors is that they want
to imitate us.  That's fine on the surface, a GPLed clone solves the
immediate problems you see, but it doesn't address how to solve the next
generation of problems.  You'd need a team of at least 6-8 senior kernel
level developers working full time for several years to get BK to the
```

https://lkml.org/lkml/2002/10/6/163

he thought his SCM was providing an irreplaceable service to the community



It costs a lot of money to do what we are doing, we know exactly how much, and a GPLed answer won't support those costs. We have to do what we are doing in order to support the kernel team and our other users. We see no other choice and not one of you have presented a viable alternative in the last 5 years.

The reason we don't want to help our competitors is that they want to imitate us. That's fine on the surface, a GPLed clone solves the immediate problems you see, but it doesn't address how to solve the next generation of problems. You'd need a team of at least 6-8 senior kernel level developers working full time for several years to get BK to the point where it won't need to be enhanced in order to support something like the kernel for the next 20 years (or more). If we GPL it or we allow clones, all that does is stop the development. It's not a question of is there the ability in the community to do what we do, there certainly is. It's a question of will they. And the answer is no they won't or they would have already. The problems that we solve aren't new at all. They just aren't that all fun to solve. Our user base is small, they are very picky, there isn't a lot of money or fun here, so why would anyone do what we do?

https://lkml.org/lkml/2002/10/6/163

You can argue all you like that I'm wrong, I'm misguided, I don't have a clue about opensource or whatever. The problem is that if I did what you'd like to see, GPL the code, and it turns out I was right, there is no turning back. That's a gamble I'm unwilling to make because I am positive of the outcome. And given what I've been doing for the last 5 years,

Keep in mind this is 2002. Github launched in 2008.

Why does Git work so well for Linux?

Torvalds: Well, it was obviously designed for our workflow, so that is part of it. I've already mentioned the whole "distributed" part many times, but it bears repeating. But it was also designed to be efficient enough for a biggish project like Linux, and **it was designed to do things that people considered "hard" before git - because those are the things *I* do every day.**

Just to pick an example: the concept of "merging" was generally considered to be something really quite painful and hard in most SCM's. You'd plan your merges, because they were big deals. That's not acceptable to me, since I commonly do tens of merges a day when in the merge window, and even then, the biggest overhead shouldn't be the merge itself, it should be testing the result. The "git" part of the merge is just a couple of seconds, it should take me much longer just to write the merge explanation message.

So git was basically designed and written for my requirements, and it shows.

https://www.linux.com/blog/10-years-git-interview-git-creator-linus-torvalds

and it was valuable stuff! very high-concept, very good implementation. but then Andrew Tridgell broke the license and Larry McCoy took home all his marbles.

SO: HOW WE GOT TO GIT

From: Linus Torvalds <torvalds@osdl.org>

Date: 2005-04-06 15:42:08

NOTE! BitKeeper isn't going away per se. Right now, the only real thing that has happened is that I've decided to not use BK mainly because I need to figure out the alternatives, and rather than continuing "things as normal", I decided to bite the bullet and just see what life without BK looks like. So far it's a gray and bleak world ;)

So don't take this to mean anything more than it is. **I'm going to be effectively off-line for a week (think of it as a normal "Linus went on a vacation" event)** and I'm just asking that people who continue to maintain BK trees at least try to also make sure that they can send me the result as (individual) patches, since I'll eventually have to merge some other way.

That "individual patches" is one of the keywords, btw. One thing that BK has been extremely good at, and that a lot of people have come to like even when they didn't use BK, is how we've been maintaining a much finer-granularity view of changes. That isn't going to go away.

https://marc.info/?l=linux-kernel&m=111280216717070&w=2

Linus posts code publicly, 2005-04-08 4:42:04

https://marc.info/?l=linux-kernel&m=111293537202443&w=2

“...if you want to play with something _really_ nasty (but also very _very_ fast), take a look at kernel.org:/pub/linux/kernel/people/torvalds/.

First one to send me the changelog tree of sparse-git (and a tool to commit and push/pull further changes) gets a gold star, and an honorable mention. I've put a hell of a lot of clues in there (*).

I've worked on it (and little else) for the last two days. Time for somebody else to tell me I'm crazy. It should be easier than it sounds. The database is designed so that you can do the equivalent of a nonmerging (ie pure superset) push/pull with just plain rsync, so replication really should be that easy (if somewhat bandwidth-intensive due to the whole-file format).

First person to profess not-understanding:

2005-04-08 8:38:39 Andrea Arcangeli

“OTOH if your git project already allows storing the data in there, that looks nice ;). I don't yet fully understand how the algorithms of the trees are meant to work (I only understand well the backing store and I tend to prefer DBMS over tree of dirs with hashes).”

Classic Linus response, 2005-04-08 14:26:08

https://marc.info/?l=linux-kernel&m=111297039107192&w=2

On Fri, 8 Apr 2005, Andrea Arcangeli wrote:

> Why not to use sql as backend instead of the tree of directories?

Because it sucks?

I can come up with millions of ways to slow things down on my own. Please come up with ways to speed things up instead.

>> After applying a patch, I can do a complete "show-diff" on the

>> kernel tree to see the effect of it in about 0.15 seconds.

> How does that work? Can you stat the entire tree in that time? I

> measure it as being higher than that.

I can indeed stat the entire tree in that time (assuming it's in memory, of course, but my kernel trees are _always_ in memory ;), but in order to do so, I have to be good at finding the names to stat.

In particular, you have to be extremely careful. You need to make sure that you don't stat anything you don't need to. We're not talking just blindly recursing the tree here, and that's exactly the point. You have to know what you're doing, but the whole point of keeping track of directory contents is that dammit, that's your whole job.

https://marc.info/?l=linux-kernel&m=111298275828636&w=2

Anybody who can't list the files they work on _instantly_ is doing something damn wrong.

"git" is really trivial, written in four days. Most of that was not actually spent coding, but thinking about the data structures.

http://localhost:8888/notebooks/Seminar/Git%20Tutorial.ipynb

“

```
Let the flames begin.

                Linus
```

<torvalds@osdl.org>
2005-04-06 15:42:08