thoughts from the red planet

Follow Nathan on

- **Twitter**
- **GitHub**
- **Blog RSS**
- **Email subscribe**

Search

Featured Posts

- Suffering-oriented programming
- History of Apache Storm and lessons learned
- Principles of Software Engineering, Part 1
- How to beat the CAP theorem
- Mimi Silbert: the greatest hacker in the world
- The mathematics behind Hadoop-based systems

Latest Posts

All Posts

Monday
Oct062014

History of Apache Storm and lessons learned

MONDAY, OCTOBER 6, 2014

Apache Storm recently became a **top-level project**, marking a huge milestone for the project and for me personally. It's crazy to think that four years ago Storm was nothing more than an idea in my head, and now it's a thriving project with a large community used by **a ton of companies**. In this post I want to look back at how Storm got to this point and the lessons I learned along the way.

The topics I will cover through Storm's history naturally follow whatever key challenges I had to deal with at those points in time. The first 25% of this post is about how Storm was conceived and initially created, so the main topics covered there are the technical issues I had to figure out to enable the project to exist. The rest of the post is about releasing Storm and establishing it as a widely used project with active user and developer communities. The main topics discussed there are marketing, communication, and community development.

Any successful project requires two things:

1. It solves a useful problem
2. You are able to convince a significant number of people that your project is the best solution to their problem

What I think many developers fail to understand is that achieving that second condition is as hard and as interesting as building the project itself. I hope this becomes apparent as you read through Storm's history.

Before Storm

Storm originated out of my work at **BackType**. At BackType we built analytics products to help businesses understand their impact on social media both historically and in realtime. Before Storm, the realtime portions of our implementation were done using a standard queues and workers approach. For example, we would write the Twitter firehose to a set of queues, and then Python workers would read those tweets and process them. Oftentimes these workers would send messages through another set of queues to another set of workers for further processing.

We were very unsatisfied with this approach. It was brittle – we had to make sure the queues and workers all stayed up – and it was very cumbersome to build apps. Most of the logic we were writing had to do with where to send/receive messages, how to serialize/deserialize messages, and so on. The actual business logic was a small portion of the codebase. Plus, it didn't feel right – the logic for one application would be spread across many workers, all of which were deployed separately. It felt like all that logic should be self-contained in one application.

The first insight

In December of 2010, I had my first big realization. That's when I came up with the idea of a "stream" as a *distributed* abstraction. Streams would be produced and processed in parallel, but they could be represented in a *single* program as a *single* abstraction. That led me to the idea of "spouts" and "bolts" – a spout produces brand new streams, and a bolt takes in streams as input and produces streams as output. They key insight was that spouts and bolts were inherently parallel, similar to how mappers and reducers are inherently parallel in Hadoop. Bolts would simply subscribe to whatever streams they need to process and indicate how the incoming stream should be partitioned to the bolt. Finally, the top-level abstraction I came up with was the "topology", a network of spouts and bolts.

I tested these abstractions against our use cases at BackType and everything fit together very nicely. I especially liked the fact that all the grunt work we were dealing with before – sending/receiving messages, serialization, deployment, etc. would be automated by these new abstractions.

Before embarking on building Storm, I wanted to validate my ideas against a wider set of use cases. So I sent out this tweet:

> *I'm working on a new kind of stream processing system. If this sounds interesting to you, ping me. I want to learn your use cases.*
>
> *— Nathan Marz (@nathanmarz)* ***December 14, 2010***

A bunch of people responded and we emailed back and forth with each other. It became clear that my abstractions were very, very sound.

I then embarked on designing Storm. I quickly hit a roadblock when trying to figure out how to pass messages between spouts and bolts. My initial thoughts were that I would mimic the queues and workers approach we were doing before and use a message broker like RabbitMQ to pass the intermediate messages. I actually spent a bunch of time diving into RabbitMQ to see how it be used for this purpose and what that would imply operationally. However, the whole idea of using message brokers for intermediate messages didn't feel right and I decided to sit on Storm until I could better think things through.

The second insight

The reason I thought I needed those intermediate brokers was to provide guarantees on the processing of data. If a bolt failed to process a message, it could replay it from whatever broker it got the message from. However, a lot of things bothered me about intermediate message brokers:

1. They were a huge, complex piece of the architecture that would have to be scaled alongside Storm.
2. They create uncomfortable situations, such as what to do when a topology is redeployed. There might still be intermediate messages on the brokers that are no longer compatible with the new version of the topology. So those messages would have to be cleaned up/ignored somehow.
3. They make fault-tolerance harder. I would have to figure out what to do not just when Storm workers went down, but also when individual brokers went down.
4. They're slow. Instead of sending messages directly between spouts and bolts, the messages go through a 3rd party, and not only that, the messages need to be persisted to disk.

I had an instinct that there should be a way to get that message processing guarantee without using intermediate message brokers. So I spent a lot of time pondering how to get that guarantee with spouts and bolts passing messages directly to one another. Without intermediate message persistence, it was implied that retries would have to come from the source (the spout). The tricky thing was that the failure of processing could happen anywhere downstream from the spout, on a completely different server, and this would have to be detected with perfect accuracy.

After a few weeks of thinking about the problem I finally had my flash of insight. I developed **an algorithm** based on random numbers and xors that would only require about 20 bytes to track each spout tuple, regardless of how much processing was triggered downstream. It's easily one of the best algorithms I ever developed and one of the few times in my career I can say I would not have come up with it without a good computer science education.

Once I figured out this algorithm, I knew I was onto something big. It massively simplified the design of the system by avoiding all the aforementioned problems, along with making things way more performant. (Amusingly, the day I figured out the algorithm I had a date with a girl I'd met recently. But I was so excited by what I'd just discovered that I was completely distracted the whole time. Needless to say, I *did not* do well with the girl.)

Building first version

Over the next 5 months, I built the first version of Storm. From the beginning I knew I wanted to open source it, so I made some key decisions in the early days with that in mind. First off, I made all of Storm's APIs in Java, but implemented Storm in Clojure. By keeping Storm's APIs 100% Java, Storm was ensured to have a very large amount of potential users. By doing the implementation in Clojure, I was able to be a lot more productive and get the project working sooner.

I also planned from the beginning to make Storm usable from non-JVM languages. Topologies are defined as **Thrift** data structures, and topologies are submitted using a Thrift API. Additionally, I designed a protocol so that spouts and bolts could be implemented in any language. Making Storm accessible from other languages makes the project accessible by more people. It makes it much easier for people to migrate to Storm, as they don't necessarily have to rewrite their existing realtime processing in Java. Instead they can port their existing code to run on Storm's multi-language API.

I was a long time Hadoop user and used my knowledge of Hadoop's design to make Storm's design better. For example, one of the most aggravating issues I dealt with from Hadoop was that in certain cases Hadoop workers would not shut down and the processes would just sit there doing nothing. Eventually these "zombie processes" would accumulate, soaking up resources and making the cluster inoperable. The core issue was that Hadoop put the burden of worker shutdown on the worker itself, and for a variety of reasons workers would sometimes fail to shut themselves down. So in Storm's design, I put the burden of worker shutdown on the same Storm daemon that started the worker in the first place. This turned out to be a lot more robust and Storm has never had issues with zombie processes.

Another problem I faced with Hadoop was if the JobTracker died for any reason, any running jobs would terminate. This was a real hair-puller when you had jobs that had been running for many days. With Storm, it was even more unacceptable to have a single point of failure like that since topologies are meant to run forever. So I designed Storm to be "process fault-tolerant": if a Storm daemon is killed and restarted it has absolutely no effect on running topologies. It makes the engineering more challenging, since you have to consider the effect of the process being kill -9'd and restarted at any point in the program, but it makes things far more robust.

A key decision I made early on in development was to assign one of our interns, **Jason Jackson**, to develop an **automated deploy for Storm** on AWS. This massively accelerated Storm's development, as it made it easy for me to test clusters of all different sizes and configurations. I really cannot emphasize enough how important this tool was, as it enabled me to iterate much, much faster.

Acquisition by Twitter

In May of 2011, BackType got into acquisition talks with Twitter. The acquisition made a lot of sense for us for a variety of reasons. Additionally, it was attractive to me because I realized I could do so much more with Storm by releasing it from within Twitter than from within BackType. Being able to make use of the Twitter brand was very compelling.

During acquisition talks I announced Storm to the world by writing **a post** on BackType's tech blog. The purpose of the post was actually just to raise our valuation in the negotiations with Twitter. And it worked: Twitter became extremely interested in the technology, and when they did their tech due-diligence on us, the entire due-diligence turned into a big demo of Storm.

The post had some surprising other effects. In the post I casually referred to Storm as "the Hadoop of realtime", and this phrase really caught on. To this day people still use it, and it even gets butchered into "realtime Hadoop" by many people. This accidental branding was really powerful and helped with adoption later on.

Open-sourcing Storm

We officially joined Twitter in July of 2011, and I immediately began planning Storm's release.

There are two ways you can go about releasing open source software. The first is to "go big", build a lot of hype for the project, and then get as much exposure as possible on release. This approach can be risky though, since if the quality isn't there or you mess up the messaging, you will alienate a huge number of people to the project on day one. That could kill any chance the project had to be successful.

The second approach is to quietly release the code and let the software slowly gain adoption. This avoids the risks of the first approach, but it has its own risk of people viewing the project as insignificant and ignoring it.

I decided to go with the first approach. I knew I had a very high quality, very useful piece of software, and through my experience with my first open source project **Cascalog**, I was confident I could get the messaging right.

Initially I planned to release Storm with a blog post, but then I came up with the idea of releasing Storm at a conference. By releasing at a conference:

1. The conference would help with marketing and promotion.
2. I would be presenting to a concentrated group of potential early adopters, who would then blog/tweet/email about it all at once, massively increasing exposure.
3. I could hype my conference session, building

> anticipation for the project and ensuring that on the day of release, there would be a lot of eyes on the project.

So releasing at a conference seemed superior in all respects. Coincidentally, I was scheduled to present at Strange Loop that September on a completely different topic. Since that was when I wanted to release Storm, I emailed Alex, the Strange Loop organizer, and changed my session to be **the release of Storm**. As you can see from the session description, I made sure to use the Twitter brand in describing Storm.

Next, I began the process of hyping Storm. In August of 2011, a little over a month before the conference, I wrote **a post** on Twitter's tech blog announcing that Storm would be released at Strange Loop. In that post I built excitement for Storm by showing a lot the details of how Storm works and by giving code examples that demonstrated Storm's elegance. The post had the effect I wanted and got people really excited.

The next day I did something which I thought was really clever. I started the mailing list for Storm:

> *If you want to be kept up to date on Storm or have questions, join the Google group* ***http://t.co/S7TJlCB***
>
> *— Nathan Marz (@nathanmarz)* ***August 5, 2011***

Here's why I think that was clever. A key issue you have to deal with to get adoption for a project is building social proof. Social proof exists in many forms: documented real-world usage of the project, Github watchers, mailing list activity, mailing list subscribers, Twitter followers, blog posts about the project, etc. If I had started the mailing list the day I released the project, then when people looked at it, it would have shown zero activity and very few subscribers. Potentially the project would be popular immediately and the mailing list would build social proof, but I had no guarantee of that.

By starting the mailing list *before release*, I was in a situation of arbitrage. If people asked questions and subscribed, then I was building social proof. If nothing happened, it didn't matter because the project wasn't released yet.

A mistake I made in those early days, which is bizarre since I was working at Twitter, was not starting a Twitter account for the project. Twitter's a great way to keep people up to date about a project as well as constantly expose people to the project (through retweets). I didn't realize I should make a Twitter account until well after release, but fortunately it didn't turn out to be that big of a deal. If I could do it again I would have started **the Twitter account** the same day I made the mailing list.

Between the time I wrote the post on Twitter's tech blog and the start of Strange Loop, I spent the majority of my time writing documentation for

Storm. This is the single most important thing I did for the project. I wrote about 12,000 words of carefully thought out documentation – tutorials, references, API docs, and so on. A lot of open source developers don't realize how crucial docs are: *people cannot use your software if they don't understand it*. Writing good documentation is painful and time-consuming, but absolutely essential.

The **moment of truth** came on September 19th, 2011. I had fun with the release. I started my talk by saying I had been debating whether to open source Storm at the beginning of my talk, starting things off with a bang, or the end of my talk, finishing on an exclamation point. I said I decided to get the best of both worlds by open sourcing Storm right in the middle of my talk. I told the audience the time of the exact middle of my talk, and told them to shout out at me if I hadn't open sourced it by that time. As soon as that moment came, the audience shouted at me and I released the project.

Everything went according to plan. The project got a huge amount of attention and got over 1000 Github watchers on the first day. The project went to **#1 on Hacker News** immediately. After my talk, I went online and answered questions on Hacker News, the mailing list, and Twitter.

The aftermath of release

Within four days Storm became the most watched Java, Scala, or Clojure project on Github. Within two weeks, **spider.io** announced they already had Storm **in production**. I thought that was incredible and a testament to the high quality of the project and docs at release.

As soon as Storm was released I started getting feedback from people using the project. In the first week I made three minor releases to address quality of life issues people were having. They were minor but I was focused on making sure everyone had the best experience possible. I also added a lot of additional logging into Storm in that first week so that when people ran into issues on the mailing list, they could provide me more information on what was going on.

I didn't anticipate how much time it would take to answer questions on the mailing list. The mailing list had a ton of activity and I was spending one to two hours a day answering questions. Part of what made the mailing list so time-consuming is how bad most people are at asking questions. It's very common to get a question like this: "I'm having a lot of tuple failures. Why??" Most of the time there's an easy fix as typically the user had something strange going on with how they configured or were using Storm. But I would have to spend a ton of time asking the user follow-up questions to get them to provide me the information they already had so I could help them. You'd be amazed at how often a user fails to tell you about something really bizarre they did, like running multiple versions of Storm at once, manually editing the files Storm daemons keep on local disk, running their own modified version of Storm, or using a shared network drive for the state of Storm daemons. These endless hours I spent on the mailing list became very draining (especially since at the same time I was building a brand new team within Twitter) and I wouldn't get relief for well over a year.

Over the next year I did a ton of talks on Storm at conferences, meetups, and companies. I believe I did over 25 Storm talks. It got to a point where I could present Storm with my eyes closed. All this speaking got Storm more and more exposure.

The marketing paid off and Storm acquired production users very quickly. I did a survey in January of 2012 and found out Storm had 10 production users, another 15 planning to have it in production soon, and another 30 companies experimenting with the technology. To have that many production users for a major piece of infrastructure in only 3 months since release was very significant.

I set up a "Powered By" page for Storm to get that last piece of critical social proof going. Rather than just have a list of companies, I requested that everyone who listed themselves on that page include a short paragraph about how they're using it. This allows people reading that page to get a sense of the variety of use cases and scales that Storm can be used for. I included a link to my email on that page for people who wanted to be listed on it. As I did the tech talk circuit, that page continued to grow and grow.

Filling the "Powered By" page for a project can be frustrating, as there can be a lot of people using your project that you're not aware of. I remember one time I got an email from one of the biggest Chinese companies in the world asking to be listed on Storm's Powered By page. They had been using Storm for over *a year* at that point, but that whole time I had no idea. To this day I don't know the best way to get people to tell you they're using your software. Besides the link to my email on Storm's Powered By page, the technique I've used is to occasionally solicit Powered By submissions via Twitter and the mailing list.

Technical evolution of Storm

Storm is a far more advanced project now than when it was released. On release it was still very much oriented towards the needs we had at BackType, as we had not yet learned the needs of larger companies for major infrastructure. Getting Storm in shape to be deployed widely within Twitter drove its development for the next 1.5 years after release.

The technical needs of a large company are different than a startup. Whereas at a startup a small team manages the entire stack, including operations and deployment, in a big company these functions are typically spread across multiple teams. One thing we learned immediately within Twitter is that people didn't want to run their own Storm clusters. They just wanted a Storm cluster they could use with someone else taking care of operations.

This implied that we needed to be able to have one large, shared cluster running many independent applications. We needed to ensure that applications could be given guarantees on how many resources they would get and make sure there was no possible way one application going haywire would affect other applications on the cluster. This is called "multi-tenancy".

We also ran into process issues. As we built out the shared cluster, we noticed that pretty much everyone was configuring their topologies to use a huge number of resources – way more than they actually needed. This was making usage of the cluster very inefficient. The problem was that no one had an incentive to optimize their topologies. People just wanted to run their stuff and have it work, so from their perspective there was no reason *not* to request a ton of resources.

I solved both these issues by developing something called the **"isolation scheduler"**. It was an incredibly simple solution that provided for multi-tenancy, created incentives for people to use resources efficiently, and allowed a single cluster to share both production and development workloads.

As more and more people used Storm within Twitter, we also discovered that people needed to monitor their topologies with metrics beyond what Storm captures by default. That led us to developing Storm's excellent **metrics API** to allow users to collect completely custom, arbitrary metrics, and send those metrics to any monitoring system.

Another big technical jump for Storm was developing Trident, a micro-batching API on top of Storm that provides exactly-once processing semantics. This enabled Storm to be applied to a lot of new use cases.

Besides all these major improvements, there were of course tons of quality of life improvements and performance enhancements along the way. All the work we were doing allowed us to do many releases of Storm – we averaged more than one release a month that first year. Doing frequent releases is incredibly important to growing a project in the early days, as each release gives you a boost of visibility from people tweeting/talking about it. It also shows people that the project is continuously improving and that if they run into issues, the project will be responsive to them.

Building developer community

The hardest part about building an open source project is building the community of developers contributing to the project. This is definitely something I struggled with.

For the first 1.5 years after release, I drove all development of Storm. All changes to the project went through me. There were pros and cons to having all development centered on me.

By controlling every detail of the project, I could ensure the project remained at a very high quality. Since I knew the project from top to bottom, I could anticipate all the ways any given change would affect the project as a whole. And since I had a vision of where the project should go, I could prevent any changes from going in that conflicted with that vision (or modify them to be consistent). I could ensure the project always had a consistent design and experience.

Unfortunately, "visionary-driven development" has a major drawback in that it makes it very hard to build an active and enthusiastic development community. First off, there's very little room for anyone to come in and make major contributions, since I am controlling everything. Second, I am a major bottleneck in all development. It became very, very hard to keep up with pull requests coming in (remember, I was also building a brand new infrastructure team within Twitter at the same time). So people would get discouraged from contributing to the project due to the incredibly slow feedback/merge cycle.

Another drawback to centering development on myself was that people viewed me as a single point of failure for the project. People brought up concerns to me of what would happen if I got hit by a bus. This concern actually limited the project less than you would think, as Storm was adopted by tons of major companies while I was at the center of development, including Yahoo!, Groupon, The Weather Channel, WebMD, Cerner, Alibaba, Baidu, Taobao, and many other companies.

Finally, the worst aspect to centering development on myself was the burden I personally felt. It's a ton of pressure and makes it hard to take a break. However, I was hesitant to expand control over project development to others because I was worried about project quality suffering. There was no way anyone else would have the deep understanding I did of the entire code base, and inevitably that would lead to changes going in with unintended consequences. However, I began to realize that this is something you have to accept when expanding a developer community. And later on I would realize this isn't as big of a deal as I thought.

Leaving Twitter

When I left Twitter in March of 2013 to pursue my current startup, I was still at the center of Storm development. After a few months it became a priority to remove myself as a bottleneck to the project. I felt that Storm would be better served with a consensus-driven development model.

I think "visionary-driven development" is best when the solution space for the project hasn't been fully explored yet. So for Storm, having me controlling all decisions as we built out multi-tenancy, custom metrics, Trident, and the major performance refactorings was a good thing. Major design issues can only be resolved well by someone with a deep understanding of the entire project.

By the time I left Twitter, we had largely figured out what the solution space for Storm looked like. That's not to say there wasn't lots of innovation still possible – Storm has had a lot of improvements since then – but those innovations weren't necessarily surprising. A lot of the work since I left Twitter has been transitioning Storm from ZeroMQ to Netty, implementing security/authentication, improving performance/scalability, improving topology visualizations, and so on. These are all awesome improvements but all of which were already anticipated as directions for improvements back in March of 2013. To put it another way, I think "visionary-driven development" is necessary when the solution space still has a lot of uncertainty in it. When the solution space is relatively well understood, the value of "visionary-driven development" diminishes dramatically. Then having that one person as a bottleneck seriously inhibits growth of the project.

About four months before leaving Twitter, Andy Feng over at Yahoo! started pushing me hard to submit Storm to Apache. At that point I had just started thinking about how to ensure the future of Storm, and Apache seemed like an interesting idea. I met with Doug Cutting, the creator of Hadoop, to get his thoughts on Apache and potentially moving Storm to Apache. Doug gave me an overview of how Apache works and was very candid about the pros and cons. He told me that the incubator could be chaotic and would most likely be painful to get through (though in reality, it turned out to be an incredibly smooth process). Doug's advice was invaluable and he really helped me understand how a consensus-driven development model works.

In consensus-driven development, at least how its done by many Apache projects, changes are voted into a project by a group of "committers". Typically all changes require at least two +1 votes and no -1 votes. That means every committer has veto power. In a consensus-driven project, not every committer will have a full understanding of the codebase. Many committers will specialize in different portions of the codebase. Over time, some of those committers will learn a greater portion of the codebase and achieve a greater understanding of how everything fits together.

When Storm first transitioned to a consensus-driven model, most of the committers had relatively limited understandings of the codebase as a whole and instead had various specialized understandings of certain areas. This was entirely due to the fact I had been so dominant in development – no one had ever been given the responsibility where they would need to learn more to make good decisions. By giving other people more authority and stepping back a bit, my hope was that others would fill that void. And that's exactly what happened.

One of my fears when moving to consensus-driven development was that the quality of changes would drop. And indeed, some of the changes that went in as we transitioned had some bugs in them. But this isn't a big deal. Because you'll get a bug report, and you can fix the problem for the next release. And if the problem is really bad, you can cut an emergency release for people to use. When I was personally making all development decisions, I would thoroughly test things myself and make use of my knowledge of the entire codebase such that anything that went out in a release was extremely high quality. But even then, my code sometimes had bugs in it and we would have to fix them in the next release. So consensus-driven development is really no different, except that changes may require a bit more iteration to iron out the issues. No software is perfect – what's important is that you have an active and responsive development community that will iterate and fix problems that arise.

Submitting to Apache

Getting back to the history of Storm, a few months after leaving Twitter I decided I wanted Storm to move to a consensus-driven development model. As I was very focused on my new startup, I also wanted Storm to have a long-term home that would give users the confidence that Storm would be a thriving project for years to come. When I considered all the options, submitting Storm to Apache seemed like far and away the best choice. Apache would give Storm a powerful brand, a strong legal foundation, and exactly the consensus-driven model that I wanted for the project.

Using what I learned from Doug Cutting, I eased the transition into Apache by identifying any legal issues beforehand that would cause problems during incubation. Storm made use of the ZeroMQ library for inter-process communication, but unfortunately the licensing of ZeroMQ was **incompatible** with Apache Foundation policy. A few developers at Yahoo! stepped up and created a replacement based on Netty (they all later became Storm committers).

In forming the intitial committer list for Storm, I chose developers from a variety of companies who had made relatively significant contributions to the project. One person who I'm super glad to have invited as a committer was Taylor Goetz, who worked at Health Market Science at the time. I was on the fence about inviting him since he hadn't contributed much code at that point. However, he was very active in the community and mailing list so I decided to take a chance on him. Once becoming a committer, Taylor took a huge amount of initiative, relieving me of many of the management burdens of the project. During incubation he handled most of the nitty-gritty stuff (like taking care of certain legal things, figuring out how to move the website over to Apache, how to get permissions for new committers, managing releases, calling for votes, etc.). Taylor later went to Hortonworks to work on Storm full-time, and he did such a good job helping shepherd Storm through the incubator that he is now the PMC chair for the project.

In September of 2013, with the help of Andy Feng at Yahoo!, I officially **proposed Storm** for incubation in Apache. Since we were well-prepared the proposal went through with only some minor modifications needed.

Apache incubation

During incubation we had to demonstrate that we could make releases, grow the user community, and expand the set of committers to the project. We never ran into any problems accomplishing any of these things. Once Storm was in incubation and I was no longer a bottleneck, development accelerated rapidly. People submitting patches got feedback faster and were encouraged to contribute more. We identified people who were making significant contributions and invited them to be committers.

Since incubation I've been just one committer like any other committer, with a vote no stronger than anyone else. I've focused my energies on any issues that affect anything core in Storm or have some sort of difficult design decision to work out. This has been a much more efficient use of my time and a huge relief compared to having to review every little change.

Storm officially graduated to a top-level project on September 17th, 2014, just short of three years after being open-sourced.

Conclusion

Building Storm and getting it to where it is now was quite a ride. I learned that building a successful project requires a lot more than just producing good code that solves an important problem. Documentation, marketing, and community development are just as important. Especially in the early days, you have to be creative and think of clever ways to get the project established. Examples of how I did that were making use of the Twitter brand, starting the mailing list a few months before release, and doing a big hyped up release to maximize exposure. Additionally, there's a lot of tedious, time-consuming work involved in building a successful project, such as writing docs, answering the never-ending questions on the mailing list, and giving talks.

One of the most amazing things for me has been seeing the huge range of industries Storm has affected. On the **Powered By page** there are applications listed in the areas of healthcare, weather, analytics, news, auctions, advertising, travel, alerting, finance, and many more. Reading that page makes the insane amount of work I've put into Storm feel worth it.

In telling this story, I have not been able to include every detail (three years is a long time, after all). So I want to finish by listing many of the people who have been important to Storm getting to the point it is at today. I am very grateful to all of these people: Chris Aniszczyk, Ashley Brown, Doug Cutting, Derek Dagit, Ted Dunning, Robert Evans, Andy Feng, Taylor Goetz, Christopher Golda, Edmund Jackson, Jason Jackson, Jeff Kaditz, Jennifer Lee, Michael Montano, Michael Noll, Adrian Petrescu, Adam Schuck, James Xu, and anyone who's ever contributed a patch to the project, deployed it in production, written about it, or given a presentation on it.

*You should follow me on Twitter **here**.*

Share Article

Reader Comments

There are no comments for this journal entry. To create a new comment, use the form below.