How to manage complexity in distributed systems

Guillem Borrell, NFQ & UC3M
guillemborrell@gmail.com
guillem.borrell@nfq.es

May 18, 2017

Two years ago I was asked to structure the development of component-based distributed applications. This was a chance to take a look at the architecture trends and to analyze how to deal with complexity. We already know that complexity can be managed with more and better abstractions, but building abstractions is one of the most complex parts of programming. This article is an opinionated analysis of the patterns that tend to produce successful abstractions, and the path that lead to them. The most relevant conclusion is that, since abstraction layers are a way to communicate with others, it is almost impossible to ignore the social aspects of programming when dealing with them.

1 A new generation of applications.

1.1 A personal background.

About three years ago I was asked to take a look at the performance issues of some applications developed at NFQ Solutions. At that time, I was finishing a PhD on wall bounded turbulence, which is a completely different topic. However, such a long and hard thesis trained me well at applyind the Scientific Method, and allowed me to use very large supercomputers. This means that, before looking at the scalability issues of an enterprise application, I already had a solid theoretical idea of what works and what does not.

When I completed my PhD, I was asked to look closely at those performance and scalability issues, and to find a solution for them. The idea was to use a scientific approach, and to use bleeding edge technologies to design a new architecture for the next generation of applications at NFQ. We called that project PALM. The meaning of the acronym is not important. It was a research project at the beginning, with

no guarantees that the outcome of my work would never be fully applied to appications deployed in production.

During my PhD I had to write small programs that were intented to solve a particular problem. The goal was to obtain an accurate result as fast as possible, and the time they spent running was seldom longer that it took to develop them. But I also had to implement very large parallel programs for supercomputers. They are completely different beasts. One of them has consumed hundreds of millions of CPU hours in supercomputers. Those large programs paid almost no attention to architecture, but a lot of effort went on performance optimization. If you spend six months developing but a simulation takes two years to complete, you run out resources. But if you spend one year developing, and the simulation takes one year to run, yo may make it. You may be surprised the amount of theses and papers published with half-cooked results. This is a useful example to understand that, in some particular cases cases, spending some time optimizing the application is a requirement. Now think about some enterprise applications that run for five years or a decade, or that are responsible to asses the financial risk of the full portfolio of a bank.

I had two years to learn about application architecture, which is quite a lot of time compared to the time you have to do some research in a middle-sized companies. But architecture is a weird topic. It has a lot to do about technology, but there's also some craftmanship. In addition, when you ask an application architect for advice, they usually tell you to repeat what they have previously done. If you know a little about how the difference between correlation and causality affects your daily life, here you have one example. If you repeat a successful strategy, there is no guarantee that you will be successful again. On the other hand, following a strategy that has already failed most likely results in yet another failure. Architecture is a complex discipline that is also quite difficult to learn, unless you are willing to start doing things by yourself, and to learn by your own mistakes.

This is why I started pylm (http://pylm.readthedocs.org/), a small library to create components and to connect them. Pylm has been relatively successful. It will be put in production by NFQ sooner rather than later in a tool for asset liability management. Pylm, and other tools developed below the umbrella of the PALM project have significantly simplified the way our proprietry applications are built now. But apart from the final product of the process, which is the pylm library, there have been mistakes and good ideas too.

1.2 Distributed applications and scalability.

Until roughly the beginning of this decade, most applications used the three-tier approach. Your application is divided between the presentation layer, the business layer, and finally, the data access layer. Now we all know that this architecture is not *Internet Scale*, whatever that means. I have a different theory about why this architecture has been abandoned forever.

With time, applications become more and more complex. New data sources, new services, multiple databases, proxies, caches for performance. Layers have become more

of a conceptual label than a functional division. We have now the frontend, that runs in the browser or on an app, and the backend. And the world has settled on the fact that the frontend and the backend talk to each other via a RESTful API.

I think that we forgot about the three-tier architecture because we realized that applications are quite organic, and as soon as you need extra performance architectural considerations are downgraded to a second level. We just assume that complexity is a byproduct of convenience.

Another important change is that we no longer integrate libraries. Now we integrate third party components. I don't think that anyone would think nowadays to integrate Lucene to her database. Probably many of you haven't even heard about Lucene. She would dump her data to Elasticsearch, even if that means relying on a third party tool, or duplicating data. Now we even push traces and logs to third party components. Everyone is running on the cloud anyways, and technically the servers are yet another third-party owned commodity. There is a strong reason for this change. If you build your application from components that scale, it is reasonable to think that the application will scale too.

The next step is to stitch all those components together. When I started asking about what to do, or looking for previous experiences in blogs and tech journals, I found mostly three kinds of answers. The first one was to use a message queue or a broker. This sounded too general to me, since brokers and message queues are very powerful tools. The second were *streaming* libraries, like Storm, Flink or Spark streaming. All these libraries looked more like yet another component than glue, and they were too opinionated. Choosing one of them also felt like doing a leap of faith, and in case of failure, it was maybe to hard to switch to a different solution. The third one was to go for a microservice architecture. Because microservices is what everyone is doing.

My decision was to go for a fourth option, that curiously was not encouraged by everyone. At that time, I had an intuition that I needed to learn more about application design and architecture. A byproduct of that decision was that, from time to time, I got people asking me why didn't I used x instead. The scientist in me is triggered everytime I get this kind of suggestion. Following a successful path again does not guarantee success for a second time. On the other hand, an unsuccessful path most likely lead to failure. You can't know if you will be right, but you can know when you are probably wrong. This is the kind of advise you get from application architects all the time, to follow their paths for success. The actual good advise is the following. Knowing a wide set of tools is important, but knowing what you are doing is even more important. And at that time I did not really know what I was doing. Like at the beginning of all research projects.

1.3 The new business logic "layer".

If tiers are dead, there is no such thing as a business logic layer anymore. The functional logic is scattered all around our components. Even worse, some logic that one day executes in a component may run within a different one some time later. I have seen business logic running within Kafka brokers. It is scattered all around, and putting all togeter within a single library is not necessarily a good idea.

What it used to be the business logic tier has become a series of modules, or packages. And here is where the Conway's law is important. The Conway's law states that if as a group you design a system, your system will be a map of your group. The social boundaries of any organization will probably become the boundaries of the system. It is a good sociological organization, and we can go for or and against this phenomenon.

My advice is to assume that this will happen, and to organize modules and packages in a way that a single developer is able to implement and document all functionalities. This has a secondary beneficial consequence, is that the developer also develops a sense of ownership and liability for that piece of software. Breaking up things into smaller pieces can cause integration issues, that can be easily solved with subpackaging a namespace package, and a proficient use of setuptools and pip. You can find in github a package called nfq, that is at the same time our namespacke package and a template for subpackage development. Subpackaging seems to work for us, and works for Google too.

1.4 A theory for not developing bad applications.

There is a little-known methodology in engineering called axiomatic design. I am an aerospatial engineer, and I know that there are quite a lot of methodologies besides waterfall and agile. Axiomatic design, like any other methodology, presents serious caveats, but it can offer some insight too. The real content of axiomatix design are its two axioms: the independence axiom, and the information axiom.

The former states that a good design maintains the independence between the functional requirements, and the latter says that a good design is the one that minimizes its information content. Note that the two axioms are mutually exclusive in practice. If you go for maximum independence, you probably end up with too much pieces to implement a given functionality.

This new breed of distributed applications, regardless of the architectural paradigm one uses to build them, can be analyzed using these two axioms. We can say that a design that requires less components with a simpler API is less complex too. This is rather obvious, and my intention is to emphasize that these ideas are not mine at all. But the most important thing is that these axioms are useful to tell if our applications are well crafted, but give no advice about how to build them.

2 Abstraction

This is probably a spoiler, but the way we deal with complexity is through abstraction. We pick a complex concept, we agree on the meaning of the concept, and then we assign a name to it. Now we can use this simbol, together with other symbols we agree on to build yet more symbols. This capability of dealing with symbols is probably what makes us unique within the animal kingdom, and we have become very good at it.

Computing has become successively simpler because we have put layers and layers of abstraction. We put pandas dataframes over numpy arrays over memory buffers over an interpreter coded in C that manages bytecode that is translated to machine

language. Now we use dataframes like a *lingua franca* for data analyisis, because symbol that is most useful for peer-to-peer communication is the one with the highest level of abstraction.

We can apply the two axioms to determine the fitness of the abstractions the following way.

Independence axiom	→	Abstractions are orthogonal and not ambiguous. They map the functional requirements.
Information axiom	→	The requirements are fulfilled with the minimum amount of abstractions possible.

Note that the two axioms are mutually exclusive. If we try to be precise, we may end with a lexicon that is too extense, while brevity and jargon usuall drives to ambiguity.

2.1 Effective abstractions and communication.

Building an API, an abstraction that has to be understood by someone else, requires certain skill. Some people are better than others at *explaining*. And anyone has a different concept of *self-explanatory*. We are undeniably the center of our world, but we are just one user, and things that are *self-explanatory* to us are not necessarily successful.

Abstractions get better the same way we get smarter. Talking to smart people. I've said that before, and the question I had immediately was, "what do I do if there isn't anyone smarter than me around?". Just talk to a frend. "What do I do of I have no friends?". Well, there's a solution even in that case.

Abstractions require domain-specific knowledge, they are hard to get right, and in consequence they are seldom bikeshedded. If you organize a meeting to talk about how to organize an API, or to design a module, it is highly probable that you get constructive feedback. It does not have to be a long meeting, sometimes grabbing a coffee and asking a coworker if what you are doing makes sense is enough. In addition, discussing abstractions is much more pleasant than discussing implementations. I don't remember an unsuccessful meeting discussing abstractions within these past two years. Some of them discussed abstractions in pylm, others discussed abstractions in python modules within the project.

I think that abstractions are seldom discussed because we get them wrong all the time, and probable outcome of the meeting is that you have to start over. But remember that this is the kind of feedback you are looking for. Since you cannot know if you are right, knowing when you are wrong as fast as possible is important. Let me give some examples to illustrate this argument.

The business logic of PALM requires to obtain the schedule of payments for several types of mortgages. There are bullet mortgages, French mortgages, German, fixed-rate, variable-rate, with and without prepayments... This goes on and on. We sat down to discuss the implementation of a library that deals with all this functional logic. One of the most senior engineers, a very brilliant individual, designed this first implementation.

We had a short meeting and it was clear that that did not work. He is brilliant and humble at the same time, and he understood immediately that we were not understanding the design. So we threw that away and went on. The second design also started cringing some months later when we identified some performance issues. Another engineer started hacking the library and found that it was just too hard to change anything without modifying several files and classes. Remember, maximum independence, minimum information. We throw that away too and we started from scratch. The engineer that identified those problems is now developing the library. The first thing she did was to sit down with a senior engineer, in this case it was me, and to discuss the new structure of files and classes. We concluded that there was nothing wrong in her schema. Again, we could not conclude that the design was right, but at least it was not obvious that it was wrong.

What if you are an entrepreneur that has no other choice than to work alone? Write documentation. The pylm library was implemented twice. After the first implementation I realized that one engineer that was writing a prototype product with the library was struggling to understand the API. But it was not until I started writing the user documentation that I understood that the implementation was a mess. The reason was that the main design hypothesis made sense at the beginning, but it turned out to be not working at all. I thouht that it was a good idea to make all the components as similar as possible, and to let them self-organize in a way. This automated several parts of the configuration, but the parts that were not automated were too difficult to understand. In any scientific work, when your initial hypotesis is wrong, you start over. It is surprising the mount of things that you realize are completely wrong while writing the documentation. But for that, your documentation must read like a book. Assuming that the whole set of docstrings make useful documentation is just wrong. Explaining what was your intention when you offered some functionality is also important. About one month ago the same engineer was able to send a patch to pylm with almost no help in a couple of hours. He is really smart, but the previous version was so broken that he needed almost a day for that.

Features need users. Otherwise it's just overengineering. No matter how cool some functionality you think it is. If noone needs it, don't add it. One of the reasons why the first version of pylm was so complicated was that it had built-in failure detection and recovery. Components were pinged all the time, and schedulers could know if some worker was faulty. But this kind of failure seldom occurred, because the application did not run in a million servers. The most usual failure was due to wrong input. This failure recovery system added lots of code and complexity, and for most of our application, understanding failure at the application level, not at the system level, was more convenient. If a feature that has no user cannot be criticized. Don't write it. It may seem great to you, but most likely you will be overengineering your application. I know sometimes is hard to overcome the urge to add some cool feature. If noone asked for it, don't put it.

2.2 Getting better (hopefully) abstractions.

Hopefully, after some time designing distributed applications, you get better at it. You build an intuition of what it usually works, and while you may still be wrong sometimes, this happens less and less often.

One of the reasons why the second implementation of pylm has been relatively successful is that contains two API with two different levels of abstraction. I realized that I needed an API to build the framework itself, because complexity was getting out of hand already during development. Now pylm has a low level API, that deals with parts, connections, protocols... And a high level API that allows a user to create components from patterns. This is the kind of designs you get when you get better at abstractions.

One of the best ideas I had during the development of PALM was something called IDN, that stands from Instrument Definition Notation. It is a file format based on JSON, with some added pragmas, that describes how a portfolio must be evaluated. It has a two-page long description of the standard, and it comes with a small interpreter and virtual machine. Abstractions are as useful as the amount of meaning they contain. A file that follows the IDN standard tells PALM how a portfolio must be evaluated, which is a decent amount of meaning. There are very few things that are able to carry as much meaning as a language. Now the word "IDN" is more or less common between the the members of the project, and this fact suggests that it has been a good abstraction. It is also useful to separate between frontend (what asks for the valuation) and the backend (what actually makes the computations), since they communicate with IDN files.

After some time into the project we also realized which kind of abstraction pylm meant. In the three-tier architecture, the application is the code that glues the different libraries together. In modern distributed applications the connections between the components are the application. We have already seen that this connection can be managed in multiple ways. A framework for streaming processing, a broker, or a microservice management system is the abstraction for the design of your application now. It gives a definition of what a component is, or a model for connections. In retrospect developing our own library was a good idea, because it settles many details about how the application works. This is the reason why I believe that many companies that started their applications using a streaming framework, or a microservice management system, they will start developing their own after some time. And I think that if they do is because they understand better the abstractions required by their application. The fact that a computational pipeline like the one provided by Storm is completely different than the flat organization of a microservice-based architecture. This is relatively obvious, and it is somewhat curious that this not so tiny detail has been unnoticed by many application architects.

2.3 Thinking about the long term.

Long term means change. People switch jobs, move to another department, change the goals in their lives... Designing future-proof systems is tremendously complex, and sometimes impossible. There are catastrophic events, black swans, that may break any

possible plan, like one key third-party library dying and going unmaintained. However, there are a set of interesting practices that may work.

It's no secret that good documentation is key. But documentation must also make an effort to go beyond the usage of the package, the module, or the application. The documentation must read like a book, and explain the reasons behind the technical decisions that have been made. The manual for the C programming language was a book that could be read from the beginning to the end. I think that Python has done this way better than other ecosystems like Java. Sphinx encourages to write books, in fact, you have to install an addon to embed docstrings in your documentation. The documentation must explain the reason for choosing one abstraction over any other possible choice, and it will be easer for another developer to pick up the work knowing that.

Once you have a plan for your application, it is better to develop your own thing that keeps that plan than steering everything to be able to use a third party tool. Regardless o the coolness of the tool. We were able to develop framework that connects general components in about a year, and yow will be able to do it too. And if you are not successful ad developing a framework, you will probably have a better plan for your application afterwards.

Do packaging right. One of the features that have made Java a successful tool is that you can pack everything in a jar file. Namespace packages, dependencies, pip, repositories... Keeping tens of packages organized is hard, and there shoud be at least one *bookeeper* that knows what each package is for.

Note that many open source projects embed those practices. Pull requests are usually an educted conversation between a developer and a maintainer. Developers get better pull request after pull request. Many open source communities have now a code of conduct precisely to improve that process.

Finally, the most important conclusion of this essay is that you get better at abstractions when you get better at communicating too. The whole team must make an effort at that. Something that is not clearly explained is as bad as something wrong. Try to avoid slides. Write short articles or handbooks, and give tutorials. You probably have heard that whiteboards are imporant, and that you should be able to give an impromptu talk in front of a whiteboard. I think that it is way better to go a step further, and be able to give an impromptu talk without a whiteboard.

The key to manage complexity is to be better at communicating.