Fail Fast vs. Robustness

Asked 14 years, 11 months ago Modified 14 years, 10 months ago

Viewed 969 times

21

Our product is a distributed system. The modules I work on are fairly new, quite rigorous, well tested. They were developed with recent best practices in mind. Other modules can be considered as legacy software.

While I'm vigilant about everything that happens within modules I'm responsible for, I'm under constant pressure to work with bad data sent to me from the other modules. At heart, I'm a "Fail Fast" principle developer and as a result , when problems arise I usually am able to eliminate the possibility of error in my modules. It's not so much about blame, just saving wasted effort in chasing bugs in the wrong places.

But the argument I keep coming up against is: "We can't let this stuff fail in production, the customer expects this to work, why don't you work around this problem". And this would be an argument for robustness: be liberal in what you accept, conservative in what you send.

I should also note that these are mostly intermittent problems. We see them in integration tests but they are hard to reproduce. Timing and concurrency are involved.

I'm having a hard time balancing between the two principles. Part of it is my worry that if I start allowing and propagating exceptional data, I'm inviting trouble and I won't have as much confidence in my system. But I can't argue against keeping the system working even if other modules are sending me wrong data. The reason other modules aren't getting fixed is that they are too complex and fragile, while mine still appear clear and safe. But if I don't resist the pressure, my modules will slowly be saddled with the same problems I've been rejecting until now.

I should say that the system is not "crashing" in production, but my module may simply display an error to the operator and ask them to contact support. A crash would be a big problem, but if I'm reporting the error clearly, then isn't this the right thing to do? I suspect that my peers just don't want the customer to see any problems, period. But my module is rejecting data from other modules within our product, not customer input. So it seems to me that we are just not tackling problems.

So, do I need to be more pragmatic or hold my ground?

performance **reliability** **robustness**

edited Jan 28, 2010 at 7:26

asked Jan 28, 2010 at 7:21

tolak
243 ● 1 ● 5

8 Answers

Sorted by: Highest score (default)

4

I share the "fail fast" preference/principle. Don't think of this as a conflict of principles though, its more a conflict of understanding. Your counterpart has some unspoken requirement ("dont show the user a bad time") that implies some missed requirement. You did not have a chance to think about/implement this requirement beforehand, so the requirement has left a bad taste in your mouth. Forget this viewpoint, re-approach it as a new project with a fixed requirement you can work against.

Maybe the best result is to give an error message like you displayed. But it sounds like you implemented it before having buy-in from your counterpart, when they had a choice to accept it. Earlier communication about what you were doing could have addressed something like that.

Be careful in how you prevent the ideas. Constantly referring to the other systems "too complex and fragile" might be rubbing people the wrong way. Express simply the systems are new to you and take longer to understand. Do put the time into understanding them, so you do not reduce peoples expectations of your capability.

Share Follow

answered Jan 28, 2010 at 7:44

Frank Schwieterman
24.5k ● 15 ● 95 ● 132

Good observation about the "missing requirement" of not displaying failure. Luckily we've only seen this in integration tests, it wasn't an error that was anticipated, e.g. a weired concurrency/timing problem, but because data handling is rigorous, it immediately flagged an issue. It was essential for us to look at the problem, and from there decide how to handle it. Only seeing these errors led us to the conclusion that we have a concurrency problem. – tolak Jan 30, 2010 at 9:12

3

I'd say that it depends on what happens if you don't halt. Does someone's paycheck get processed wrong? Does the wrong order get sent out? That would be worth stopping for.

If possible, have your cake and eat it too - don't report the error to the user, get the customer to agree to send diagnostic reports and report every failure back. Bug the developer(s) who own the faulting module(s) to fix them. And by bug I mean file a bug against them. Or, if management doesn't think it's worth the cost of fixing, don't.

I'd also write up unit tests against those modules that fail, especially if you can tell what the original input was that caused them to generate the wrong output.

What it really comes down to though is what the person who reviews your performance wants from you, especially after you explain the problem to them, via email.

Share Follow

answered Jan 28, 2010 at 7:30

Mark Allen

1,215 ● 1 ● 15 ● 17

3

I won't get into the reasons, but you are right.

In my experience, PHB's are missing the part of the brain required to understand why fail fast has merit and "robustness" as defined by do-whatever-it-takes-eat-errors-if-necessary is a bad idea. It is hopeless. They just don't have the hardware to grok it. They tend to say things "ok you make a good point but what about the user" - it's just their version of think of the children, and signals the end of a conversion with me anytime it's brought up.

My advice is to stand your ground. Eternally.

Share Follow

answered Jan 28, 2010 at 7:39

Terry Mahaffey

12k ● 1 ● 38 ● 44

So we develop software, and don't consider what the user wants or needs? er, no, (-1) – DanSingerman Jan 28, 2010 at 8:19

4 @Dan, I didn't want to get into the core of the debate, but if you're unaware - people who argue fail fast are considering the users wants and needs; and the users wants and needs will be better served by fixing the real bug (easily identified if you fail fast) than hiding it with "robustness". This is the point that most PHBs miss. – Terry Mahaffey Jan 28, 2010 at 8:40

2

Simply put, this sounds like a "don't check for something you can't handle". The fact that you're catching the error and able to report it means you're not propagating it. But it also means that since you can report it, you have some mechanism to trap the error and, therefore potentially handle it yourself, and correct it rather than report it.

Mind, I'm assuming that your error report is more interesting than a random exception you caught some place deep in the system. But even then, if it's an exception you're testing for and you're creating (i.e. you check if the denominator is zero and send an error rather than simply inadvertently dividing by zero and catching the exception higher up), then that suggests you may well have a way of correcting the problem.

Bottom line, you need both. You need to try to make the data as error free as practical, but also report the unexpected.

I don't think that you can lock the door and cross your arms saying "it's not my problem". The fact that it's coming from "old, fragile systems" is meaningless. YOUR code is not old a fragile and clearly the efficient place, in

terms of the entire integrated system, to "fix" the data, once you've detected the problem. Yea the old modules will continue to GIGO to other, lesser systems, but those legacy modules combined with your new module are a cohesive whole and thus make up "the system".

The typical real problem here is simply the time/value equation of writing all this fix up code vs new features. That's a different debate. But if you have time, and you know things that you can do to clean up incoming data, "be liberal in what you accept" is sound policy.

Share Follow

answered Jan 28, 2010 at 7:37

Will Hartung

118k ● 20 ● 133 ● 207

Thank you for a balanced and well considered response. It helped me framed the issue. – tolak Jan 30, 2010 at 9:00

1

Thanks everyone. The case that prompted this question ended well, and partly thanks to insights I got from the answers above.

My initial reaction was to stick to fail fast, but I thought about this some more, and had reached the conclusion that one of the roles of my module is to provide a stabilizing anchor to the rest of the system. That does not necessarily mean accepting bad data, but surfacing problems, isolating them and handling them in a transparent manner until we find a solution.

I planned adding a new handler and code path for this case, which would properly execute as if it was a special use case that was previously undocumented.

We had a discussion where I reiterated the need to deal with the problem at the boundary, but was also willing to help. I outlined my plan to the other side, because I had a suspicion that my position was viewed as overly pedantic, and that the solution was perceived as me only having to turn off spurious validation of harmless data, even if it was incorrect. In reality though, the way I work is largely data driven, so I explained why it has to be correct and how behavior is driven by it and how in accommodating this data I will be implementing a special code path.

I think this gave weight to my position and it led to a more thorough discussion of the other side's aversion to fixing the data. It turned out that it was more of a weariness of dealing with an error prone legacy system than an actual obstacle. There was a relatively simple solution, it was just scary to make a change, a mindset that's fairly entrenched.

But having aired all challenges and possible solutions, we eventually agreed to fix the data, and so far it seems to have solved our problem. Our integration tests are now passing consistently, but we have also added logging and will continue to monitor it.

In summary, I think that for me, the synthesis of both principles is that fail fast is essential for surfacing problems. But once they do surface, robustness means

providing a transparent path to continue operation in a way that does not compromise the system. I was able to offer that, and by doing so, won some goodwill from the other side and got the data fixed in the end.

Again, thanks to everyone that responded. I'm too new to rate comments, but I do appreciate all the perspectives presented.

Share Follow

answered Jan 30, 2010 at 8:36

tolak

243 ● 1 ● 5

> Wow, that whole thread, including the comments and answers, was totally professional and well-though out. No knee-jerk responses, no finger pointing, no complaining. I am impressed by everyone who participated. Up vote for providing the conclusion -- though a little iffy about accepting your own answer. – MJB Apr 7, 2010 at 13:08

> I wasn't sure what to do about the answers frankly. I thought that was meant to be the course of action taken. Still learning how to work this. – tolak Apr 27, 2010 at 5:25

0

That's a tricky one. If your module receives bad data and it's "ok" for you to just do nothing with them and return, then I would suggest to write to an error log instead of showing an error to the user.

Share Follow

answered Jan 28, 2010 at 7:32

Warpin

7,053 ● 12 ● 53 ● 78

0

It kind of depends on the class of error you are getting. If the way the system is breaking means you can keep going without feeding bad data to any other parts of the system, you should do everything in your power to work with whatever input is given.

To my mind though data purity trumps working systems, you cannot allow bad data to propagate elsewhere and corrupt other systems. To the extent you can massage data to be correct and then keep going, you should do so on the theory that the data is safe and you must keep the system running...

I like to think of things in terms of data streams. Passing bad data along is polluting the whole stream, and that is bad because just like real pollution a drop can spoil a whole river of data (if one element is bad, what else can you trust?). But equally bad is blocking the flow, letting nothing pass because you spotted something you could easily remove. Filter it out and if everyone at every stage is also filter, you get clear clean data out the other end even if a few impurities started up in the middle.

Share Follow

answered Jan 28, 2010 at 8:11

Kendall Helmstetter Gelner

75k ● 26 ● 131 ● 151

0

The question from your peers is: "why don't you work around this problem"

You say that it's possible for you detect the bad data, and report an error to the user. This is the normal approach - once you know the data coming to your functions is bad, you should fail fast (and this is the recommendation from the other answers I have read here).

However, your question doesn't specify the domain in which your software is operating. If you know the data coming in is erroneous, is it possible for you to request that data again? Is it actually possible to recover from the situation?

I mentioned that the "domain" here is important. So if you have an app which displays streamed video data for example, and maybe your wireless signal is weak so the stream is corrupt, should the system "fail fast" and display an error message? Or should a poorer image be displayed, and an attempt to reconnect made if needed, depending on the magnitude of the problem?

Depending on your domain, it may be possible for you to detect bad data, and make a second request for the data without inconveniencing the user. (This is clearly only relevant in cases where you'd expect the data to be better the second time, but you do say the issues you are experiencing are intermittent and possible concurrency related)...

So, fail-fast is good, and is definitely something you should do if you can't recover. And you should definitely not propagate bad data. But if you can recover, which in some domains you can, then failing straight away is not necessarily the best thing to do.

Share Follow

edited Jan 28, 2010 at 9:03

answered Jan 28, 2010 at 8:40

David_001

5,792 ● 4 ● 32 ● 55

It's a critical system, but it is a guidance to human operators. It cannot tell an operator who has performed something straightforward that due to some internal issue, he must stop. I can understand the reluctance to surface problems in production where the user is performing correctly but the system is failing. – tolak Jan 30, 2010 at 9:05