Note: Each “thread” of discussion on this post has a page of its own. The link to this discussion is available here (VLE access is required).

Initial Post

On April 9 2014, the USA's 911 service experienced an outage of approximately 6 hours across 7 states, resulting in 11 million people being unable to use the service, and over 6600 calls not being answered (Federal Communications Commission, 2014).

The cause of this outage was due to a software flaw- it had a limit for how many unique identifiers could be assigned to calls. This limit was 40 million (Fung, 2014), and once it was reached, new database entries for incoming calls could not be created, making it impossible for them to be assigned to a call center, leading to calls being dropped. An early warning for this was present in the form of logs, however, logs related to the outage were classified as low severity, and staff on call were not able to determine that these logs could indicate an outage. A backup system existed which would have enabled rerouting, however, as the main logs were classified as low severity, the system did not activate the backup functionality (Federal Communications Commision, 2014). The report by the Federal Communications Commission also noted that the logging of this issue at a low level was a default setting which was approved by a Systems Administrator.

This example shows the importance of managing program logic when there's a chance that a set limit can be reached. This issue could also have been caught through code reviewing. This example furthermore shows the importance of validating information systems holistically. In complex systems, errors need to be viewed in the context of the entire system because of how they can cascade through systems and cause larger problems. If this were done, the Systems Administrator would have understood the severity of the logs and assigned them a higher priority, ensuring that the backup system would activate.

References

Federal Communications Commision. (2014) April 2014 Multistate 911 Outage: Cause and Impact. Available from: https://docs.fcc.gov/public/attachments/DOC-330012A1.pdf [Accessed 9 May 2021].

Fung, B. (2014) How a dumb software glitch kept thousands from reaching 911. Available from: https://www.washingtonpost.com/news/the-switch/wp/2014/10/20/how-a-dumb-software-glitch-kept-6600-calls-from-getting-to-911/ [Accessed 9 May 2021].

Response: Beran Necat

Hi Shan,

Thanks for your initial post. Despite there being a higher awareness by organisations of the risks that can occur and the impact they have, organisations still fail to address this sufficiently.

What risk management practices are present in your place of work etc..

Regards, Beran

Reply: Shan Swanlow

Hi Professor,

Thank you for the response. I currently work on software that is for a physical product used in retail stores, which is updated on a regular basis. The product updates its software via the internet, however, if the update does not work on a product in store, the product would stop working entirely and we'd have to send a technician to the store to physically resolve the issue- this is a heavy expense in terms of time and money, in addition to the damage that would be dealt to the relationships with our clients.

For this reason, risk management is something that has to be prioritized consistently, and in my experience, that is achieved by being patient and thorough with development. During development, we hold regular meetings with the aim of clarifying functionality and ensuring that each developer is aware of how their development will influence the other developers. Pair programming is also offered if anyone requires it. At the end of development, code reviews are held, with a focus on ensuring that the theory behind the implementation is correct- this is done by checking algorithm logic, edge cases, and so on. After this, extensive testing is done through a focus on integration testing (i.e. ensuring the updated code works with all the embedded hardware). Thereafter, units are upgraded in a staged manner- we upgrade a few that are easy for technicians to access, monitor the units to see if the update is stable, then continue with a mass rollout if the first wave is stable. This approach allows us to have multiple opportunities to catch errors before release, minimizes all damage and risk for the worst possible case (i.e. discovering an issue in-field), and to date, this procedure has allowed upgrades to be done safely and easily with zero issues during deployment.

Response: Michael Justus

Hi Shan,

This failure case you identified is very interesting from two aspects: (1) the limit on unique identifiers, (2) the classification level of issues.

For (1), why do you think there would be such a small limit given that in software, data types exist such as integer and Guid which allow for unique identifiers greater than 40 million. Since it was a software flaw, perhaps this limit was the result of poor requirements or undersight that no more than 40 million identifiers would ever be required; after all, the population was not as large as the present day.

For (2) you mention "managing program logic when there's a chance that a set limit can be reached". Could you explain a little more about managing program logic? I ask because program logic can be viewed as a business process, the thing that turns data into information or acts upon the data in a meaningful way. These processes surely need to be tested, validated against requirements and, as you mentioned, code reviewed.

Reply: Shan Swanlow

Hi Michael,

I agree with your suggestion that the cause may have been an undersight. The referenced report unfortunately does not give an exact time of when the software was created but it does mention that it's a part of a legacy approach (Federal Communications Commission, 2014). Underestimation as part of legacy systems isn't an uncommon issue- the internet itself currently has the same issue because it was initially thought that 4.3 billion IPv4 addresses would be sufficient, but we're running out of addresses to use (Internet Society, N.D.). For these reasons, it would make sense for the call limit to be an arbitrary value, especially considering that it could be considered "overengineering" to try and forecast when numeric limits may be reached for a system. What are your thoughts on planning for limits, and balancing realism with prediction in the process?

With regards to point 2, I mean to say that all logical branches of some behavior need to be accounted for, ideally during planning. For this case study, there's a known limit, and some action needs to be taken if the limit is or isn't reached. Even if it's unlikely, it's worth considering what to do if the limit is reached, and how to get this information, in order to guarantee longevity. I personally consider program logic to be an implementation detail- lots of lower-level details may change based on the language, but the system's functional goals would be identical. If Python 3 were used for instance, this bug would never have occurred because it has no limit on how large an integer can be (Python Software Foundation, 2012), but the system would still work according to specification. That said, your recommendation of "validating processes against requirements" is a really good way of preventing this type of issue and catching any other hard-to-predict issues that may arise from using a specific technology stack. By classifying program logic as a business process, do you think it would influence the design of an information system, as opposed to keeping it as a technical process?

References

Federal Communications Commision. (2014) April 2014 Multistate 911 Outage: Cause and Impact. Available from: https://docs.fcc.gov/public/attachments/DOC-330012A1.pdf [Accessed 16 May 2021].

Internet Society. (N.D.) Frequently Asked Questions (FAQ) on IPv6 adoption and IPv4 exhaustion Available from: https://www.internetsociety.org/deploy360/ipv6/faq/#1 [Accessed 16 May 2021].

Python Software Foundation. (2012) What's New In Python 3.0. Available from: https://docs.python.org/3.1/whatsnew/3.0.html#integers [Accessed 16 May 2021].

Reply: Michael Justus

Hi Shan,

First I want to say, wow, smarter every day. Thank you for the information regarding integer sizes in Python 3. I am impressed there is no upper limit to integer values in Python, something I am not used to in languages such as C#.

However, it feels that your statement "If Python 3 were used for instance, this bug would never have occurred" optimistically covers a hidden system bug. Why? Because the system failed at 40 million calls, which seems like it was built with an assumption, one you also wonderfully highlight in regards to IPv4.

If the designers of software are to make assumptions, surely they need to base those assumptions on tangible or measurable quantities? For example, the 40 million limits ought to have reasonably been set at say, 300 million according to the measurable quantity available at the time (PopulationPyramid, 2019) of system design. So it can be argued that instead of relying on arbitrary value for limits, software engineers or architects, ought to utilise known quantities, limits or expectations when developing their solutions.

In this regard, planning for limits cannot be unrealistic, neither can it be anything that tends toward overengineering, as you highlighted. So what is the solution? I would say as professionals, one attempts to minimise unknowns. An approach is to manage expectations for the art of the possible, use all data available to set limits and then communicate these to management, who through experience and responsibility, will agree or deny the limits. I think data is our enabler and the more we have, the better our choices in designing long-lived systems.

Classifying program logic as a business process influences the design of an information system because the two concepts are tightly coupled to each other. A business process is an abstract definition of how to achieve some business-oriented goal in a system, while program logic is the physical manifestation of those steps needed to reach that goal. And it may take one or more interconnected units of program logic (functions/methods/procedures) to accomplish a single business process.

For example "Save Customer Record" is a business process that may be defined as "Take customer data gathered from an input form and persist it into SQL Server" while the manifestation is a program logic function something along the lines of (pseudo-code):

```
FUNCTION SaveCustomer(Customer: record)
 CONNECT to SqlServer
 OPEN DB Table
 WRITE Record to Table
 RETURN Success
END FUNCTION
```

So how does classifying program logic as a business process influence design? (1) A change in one affects the other; (2) program logic may not have the required capability to fulfil business processes which could require a change in business process flow; (3) timing/security/data requirements of business processes can also influence the design of program logic.

What are your thoughts? How do you view the link between business process and program logic,

References

PopulationPyramid (2019). Population Pyramids of the World. Available from https://www.populationpyramid.net/united-states-of-america/2014/ [Accessed on 16 May 2021]