Distributed Systems Basics – Handling Failure: Fault Tolerance and Monitoring

< Prev. Next >

November 13, 2011 technology 2 Comments

Since earlier this summer I have been working on a book chapter for the Architecture of Open Source Applications text book. It is a pretty cool project because there are a lot of great contributors, and all of the profit made from text book sales goes to Amnesty International.

My chapter assignment was Distributed Systems, which was pretty broad, so I focused my writing on the architecture of large scale internet applications. Like most writing though, it is always best to cut down things, and so part of my chapter that was cut was all about handling failures particularly my sections on monitoring and fault tolerance. Since I think that these are pretty important, I posted the cut parts here and plan to link to them from my chapter. So here is a preview of my chapter (well the part that didn't make it in) below 🙂

Handling Failures

When it comes to failures, most fall into one of two buckets: hardware or software related.

- Hardware failures used to be more common, but with all of the recent innovations in hardware design and manufacturing they tend to be fewer and far between with most of these physical failures tending to be network or drive related.
- Software failures, on the other hand, come in many more varieties. And software bugs in distributed systems can be difficult to replicate and, consequently, fix.

In small, self-contained systems it is much easier to simulate the conditions required to replicate and debug issues, with most of these issues classified as being a Bohrbug, that is a bug "that manifests itself consistently under a well-defined (but possibly unknown) set of conditions" [3]. However, in more complex systems or production environments having many servers, it can be extremely difficult to find and diagnose more unusual bugs; like the Heisenbug "that disappears or alters its characteristics when an attempt is made to study it" [3].

With more hardware the probability goes up that there will be a failure somewhere. Add more software and the complex interactions between different programs creates greater chance for more bugs, including the unusual ones. As a result, any distributed design will carefully consider failure and diagnostic scenarios.

When designing distributed systems it is said that the following (perhaps normal) assumptions should be considered false (and these are so well known that they commonly referred to as the Fallacies of Distributed Computing):

1. The network is reliable.
2. Latency is zero.
3. Bandwidth is infinite.
4. The network is secure.
5. Topology doesn't change.
6. There is one administrator.
7. Transport cost is zero.
8. The network is homogeneous. [4]

my greatest hits
Looking for a place to start? Here are some of my most popular posts:

My Swan Story - a journey of self improvement, discovery, and success

The Paradox of Autonomy and Recognition

Lean Software Development – build v1s and v2s

Distributed Systems Basics – Handling Failure: Fault Tolerance and Monitoring

What Every Programmer Should Know About SEO

topics

being awesome
business
career
communication
entrepreneur
good stuff
personal
productivity
rants
self improvement
technology

By challenging each of these assumptions and looking at the system design within that context it can help identify potential risk areas. Systems that exhibit the key principles, like reliability and availability, have designs that take each of these fallacies into consideration. Handling failures in these areas can be done in an automated way, or other times the design may simply involve a plan or reaction for when they occur. And, there many different techniques that come in handy guarding against failure, redundancy was covered above, but two other important techniques are fault tolerance and monitoring.

Fault Tolerance:

Another important part of service based architectures is to set up each service to be fault tolerant, such that in the event one of its dependencies are unavailable or return an error, it is able to handle those cases and degrade gracefully. There are many methods for achieving fault tolerance in a distributed system, for example: redundancy (as described above), standbys, feature flags, and asynchrony.

Standbys – a standby is exactly that, a redundant set of functionality or data waiting on standby that may be swapped to replace another failing instance. Replication can be utilized to maintain real time copies of the master database so that data may be replaced without loss or disruption.

Feature flags – a feature flag is used to enable or disable functionality in a production system. In the event of a failure for a particular system, features that depend on that system can be turned off and made unavailable until that system comes back online.

Asynchrony – this is probably one of the more important design considerations in any distributed application. It essentially means that each service, or functional piece of the system, communicates with each of its external dependencies asynchronously, so that slow or unavailable services do not directly impact the primary functioning of the application. This also typically implies that operations aren't tightly coupled, requiring the success of one operation for another to succeed, like a transaction, and don't require services to be available to handle requests. For example, writing an image could return success to the client, even if all the copies of the file haven't been created and written to the file store. Some of the pieces of the "file write" are asynchronous – there will be a file upload, converting the file to the desired format/size, writing the converted file to disk, and then replicating it to its redundant copy. While all of those are required for the file write to be considered a success, each of those **eventually** have to occur, meaning that the client can receive "success" once the file is uploaded and the rest of the operations can happen asynchronously afterward. This means that if one of the down stream services was unavailable or congested, the other parts of the write could continue operating as expected (in this case, accepting files from clients).

Monitoring:

Extensive monitoring and logging is essential to any complex distributed system. Having many services each with a different purpose, yet still interacting with one another, can lead to highly unusual bugs when they occur. It can be hard to tell where the problem lies and where the issue needs to be resolved. One of the best ways to mitigate this confusion and help diagnose problems quickly is to be sure that all system interfaces and APIs are monitored.

However, monitoring in large-scale web systems can be challenging.

Separation and services adds complexity.

As noted earlier, a key part of scaling systems is to break up the pieces of the system into services, but because each of these services is independent of one another it can make problems harder to diagnose. There are many different points of control and they don't necessarily operate in sync with one another, making traditional sequential monitoring, or tracing the pattern of execution (like a debugger would with breakpoints in a program) much more complicated.

Furthermore, with most of these systems the communication between them can be delayed and complicated through mechanisms like retries (which is what happens when one service makes a request, like fetching an

image, and the other responds with an error, such as being too busy to serve the request, and the requesting service retries the request at a later time), which only compounds the problem of tracing sequential events.

reasons why many software systems have a debug (or noisy) mode and a production mode with less logging and information. Monitoring may or may not live on the same physical hardware, and in the event that it doesn't, there is additional communication required for the monitoring systems to track and record metrics. This extra step is another layer of interaction, which can add more complexity in understanding these systems.

Despite all the obstacles with monitoring though, extensive metrics are a key part to understanding and diagnosing problems. Here are some best practices in distributed system monitoring:

Monitor end-to-end functionality. End-to-end monitoring typically consists of an operation, or path from start to finish within the system. So for the image hosting example, this would be uploading an image, and ensuring that the image is written as expected and can be retrieved from its storage location by the client. It is the complete use case for the web system. Of times there are more than one of these cases and each one should be monitored independently.

Monitor each service independently of one another. Since each service has its own focus of control, making sure that each one is monitored is a key part to recognizing or diagnosing problems within that service. The monitoring for each service may not be the same, but often there is some overlap in monitored metrics (like standard system metrics, i.e. CPU usage, memory usage, disk and network i/o etc.). In the image hosting example, some of the service specific monitoring would be: the speed of reads, how many concurrent reads were happening for the Image Retrieval Service, whereas the Image Write Service would watch the write queue, number of connections.

Monitor "to the glass" metrics – look at things from the end users perspective. An important part of any large web system is to really understand what the end user's experience is with the web site. It is not enough to understand the internal workings, but it is important to also monitor and track the overall experience from the client. Typically this is done using an external service that will in its simplest form ping the site to ensure that it is up, and in more complex cases actually execute end-to-end use cases. This sort of metric can help diagnose problems that occur somewhere between the client and the website, and can be one of the fastest ways to uncover network problems. For global user bases, this sort of monitoring may be geographically distributed such that it will take into account the different networks.

Monitor at a frequency to detect issues before they impact customers (this also means having enough time to address the issue, so also understanding the rate of change and next steps to address it). Another key part of any sort of monitoring is ensuring that the data is sampled frequently enough to detect problems, and raise alerts before it becomes an issue. For example, if there are too many requests to upload images simultaneously but throughput is only logged every few minutes, it may not be frequent enough to detect these peak usage periods that are causing problems.

Establish a baseline on historical performance. In order to really make the most of the metrics, one must understand what is "normal" for the system. Without a baseline or clear history it is really difficult to determine if something is wrong and should be investigated. For example, problems caused from too much load or traffic on the system would be very hard to diagnose if there was not an easy way to understand what type of load or throughput was typical. Therefore it is not just enough to track metrics, but to also look at trends over time.

Monitor the monitoring systems. In addition to having monitoring systems keeping track of events within the system, it is also key to make sure that the monitoring is up and reporting as expected (otherwise there is no way to know that there is even a problem!). Most of the time it is sufficient to have something as simple as an external ping to ensure the monitoring software is responsive (and most have web front ends that make this easy.

While extensive monitoring is not necessarily required, thinking about these things ahead of time will certainly help ensure more resiliency and faster recovery in the event of failure. Monitoring is a central theme for any large scale website, and the bigger more complicated the architecture, the more important it is to have

and application. In general, application specific metrics are the most important, but can also be the hardest to measure since they are specific to the business or application and will almost always require some customization.

In most of these websites, there will be a separate system (that could be one or more hosts) that is tasked with collecting and aggregating data across multiple machines. This can be done by installing agents or instrumenting each of the machines "being monitored" and one interesting facet of these distributed systems (particularly ones in the cloud) where machines are often in flux (going up and down or becoming unavailable) is to setup a bootstrapping process that includes adding the monitoring to those new hosts; and of course doing this in an automated way.

Another challenge is choosing what data to collect and at what frequency and sampling rate makes sense for the architecture, because it is very easy to collect too much data and miss the forest through the trees (or in some cases, simply overwhelm the server). Some big websites, such as Wikipedia and Twitter [7], use software like Ganglia that will automatically aggregate data storing it in decreasing resolution as it ages; so for recent events there will be more data at a higher granularity than events in the past.

There are many great open source options for monitoring, logging and tracking events in web systems, Nagios, Zabbix, Flume, and Munin are all popular choices; and it is not uncommon to use more than one for the same system.

Having covered some of the core considerations in designing distributed systems, let's now talk about the hard part – scaling access to the data. {the rest is available in the chapter!}

[3] Unusual software bug, http://en.wikipedia.org/wiki/Unusual_software_bug
[4] Fallacies of Distributed Computing Explained, http://www.rgoarchitects.com/Files/fallacies.pdf
[7] Ganglia documentation and users, http://ganglia.sourceforge.net/

Related Posts:

- Assessing Technical Risks for Startups – New Tech Leader Series
- Understanding Company Strategy – New Tech Leader Series
- 20 Things Every iOS And Mobile Dev Should Consider in Their App Promotion Strategy
- What I loved about Surge Con 2011
- Selecting Vendors – is a startup or established company better?

Comments

Dmitry 22nd March 2012 at 10:20

Thank you Kate, good considerations ….
A spoon of constructive criticism:
– while all activities/practices, you've mentioned, are essential for managing and operating applications and systems, they are not necessarily specific to distributed systems. Even if you have one server, you still need to do all of these …

It would be nice if you could expand the topic to cover such areas as different approaches to load balancing, parallel processing, may be even multi-tenancy, which are more distinct attributes of distributed systems

kate 23rd March 2012 at 17:24

That is definitely great feedback. If I ever get to write more on the topic I would love to cover it. Mostly this post was an excerpt that was cut out of the larger chapter on distributed websites. You could really write a whole book on just monitoring in these systems!
🙂