the Availability Digest

www.availabilitydigest.com

Load Shedding

December 2012

Paul Green
Senior Technical Consultant
Stratus Technologies, Inc.

Dr. Bill Highleyman
Availability Digest

A recent thread in our LinkedIn Continuous Availability Forum covered a very important topic – load shedding.[1] What do you do if your system approaches full capacity? What do you do in an active/active system if you lose one node and the surviving nodes must carry the full load? What do you do following a failover if your backup system is smaller than your production system?

If you want to maintain a reasonable level of service, you may have to shed some of the load that is being carried by the system. But which load?

One of us (Green) started the discussion by posing the following question:

What is the appropriate load-shedding policy when a continuously-available system becomes overloaded?

All processing systems, whether manual or automated, have an upper limit on the number of transactions they can process per second. As designers of such systems, we project the capacity needed over a specific period of time and then engineer the system so that it satisfies those requirements. As the requirements change over time, we re-engineer the system to meet the new projections.

But unanticipated situations happen, whether from a natural event (a hurricane), human error, partial failure of the system itself, or simply a sudden and unanticipated increase in demand beyond the capacity of the system. Therefore, it seems that we need to specifically design for the case in which the incoming transaction rate exceeds the processing capacity of the system.

While internal queuing can handle momentary spikes, it can't handle prolonged spans of time. What policy should the system follow when it is overloaded? Should it simply let the requests queue up externally? Should it deny some requests and accept others? Some of these choices depend upon the nature of the application. All of them have impacts on society. All of them have unintended consequences.

[1] http://www.linkedin.com/groupItem?view=&gid=2586333&type=member&item=185759401&qid=3770f805-5895-4054-b4d0-a7eab6fa4fc5&trk=group_most_popular-0-b-ttl&goback=%2Egmp_2586333

In my experience, many organizations simply side-step this issue by trying to always have enough capacity. Fine, but as engineers, don't we owe it to our clients to be aware of this problem and try to solve it anyway? I can assure you that it happens with enough frequency to warrant thoughtful consideration.

We had dozens of meaningful comments on the subject. An important point that was made was that an overloaded system that cannot provide useful response times is a failing system:

From an end user's point of view, any overload situation is just another system outage. The person (or machine) waiting for a service and not receiving it could not care less whether some hardware is broken, some software ran into the woods, or there is "just" too much demand on the system. A system that is not sufficiently configured to support the current incoming load is just no longer a continuously available system.

But “failures” due to degraded response times do happen in the real world. Arguably, overloaded situations are more common than total application outages.

Another commentator noted that load shedding is not just a technical consideration:

While it is tempting to consider load shedding a technical decision, in reality it is a policy decision. Whatever the decision about how to shed load (either in response to capacity shortfalls or to a security incident), it needs to be along the lines of policies already decided at the business management level. Cut-offs often have legal and responsibility implications, quantification and management of which is a senior management responsibility.

It's a senior management decision whether they want their system to be continuously available under high load conditions or not. It can be a clear business decision to let high load bring the business down. But this should be a conscious decision, not leading to surprised faces and panicking when this actually happens.

Some suggested that there be algorithms to discard less important transactions so that others could complete.

I suggest that the app throw away or reject the transactions that it knows that it won't be able to get to. A queue scanner, for example, could notate and drop or reject transactions that it knows won't be able to finish. That way it could keep the queue 'trimmed' and perhaps give the transaction the opportunity to quickly resubmit itself to another processor.

It may be possible to put priority on incoming requests (i.e. "looking" vs. "booking" requests). If so, reject low-priority requests with a clear error message: "system busy, please retry later". I know this text sounds lame, but it seems better than just dump timeouts for all requests.

Because an overload is very unlikely, the thing we did was [to] think about priorities. The critical online application has the highest priorities and the non-critical online and batch-applications have significantly lower priorities. In addition our monitoring checks for processes causing too much load, and it reduces the priority for such processes automatically.

I'll relate one design element that we had in a law enforcement message switching application. That system processed many different types of inquiries - some critical, some routine. We designed the initial message parser to reject all messages of a given type if system resources were being taxed. The user defined the order in which we would start rejecting traffic. That let the user continue to process the higher priority workload during abnormal conditions. We quickly learned that we had to add a delay to the "reject" message to prevent the users from immediately resending and taxing the system even more.

I think the lesson learned here is that you have to plan carefully what load to shed. Unfortunately, there are many monitors on the NonStop that are themselves very heavy consumers and probably should be curtailed or reduced in capability. I would even go so far as to say that ongoing Measure (a good practice) should have data amounts reduced and intervals lengthened during your "shed."

One person thought that moving excess load to a cloud may be a possible solution:

One argument for cloud computing is the theory that there should always be enough capacity available in the background to match the current demand. There is no reason why a continuously available system could not be part of a cloud, or even that continuously available systems might be the stuff that the cloud is built of.

In the NonStop world, there is a mechanism called Persistent Cloud Services that can be used to provision such extra capacity for high load situations. But care needs to be taken - when moving load into the cloud, you'll see extra latency (potentially disturbing); and you may also get hurt if you don't have fully dedicated physical servers assigned to you, but rather rely on virtual servers which in turn might run on physical servers that get overloaded. If you have a big business peak shortly before Christmas, you might not be the only one - and the cloud you are relying on just might get overloaded too. The cloud-service providing business is a pretty tough one, margins are low, and those who do actually believe that their provider will generously invest in ample capacity not needed for the rest of the year may be in for a surprise.

However, I am dubious about cloud solutions, too risky and out of our control.

Capacity planning is more than CPU workload:

Very often, capacity planning is done by looking at the CPU load only. But we have to look at everything that could reach the limits. If your communication lines do not have the necessary bandwidth, you will never have a chance to differentiate between the messages. And problems with line capacity can sometimes have very mysterious symptoms. A few years ago, we had protocol errors on a leased line; and that was the result of a bandwidth problem. The bandwidth was increased and the problems were gone.

Or think of such things as HSMs (Hardware Security Managers). Today we need a lot of HSM capacity as we have to regard rules like PCI/DSS. What happens if an HSM goes down? Will the remaining HSMs be able to handle the load?

So good and effective capacity planning involves all the components needed. I still prefer to look at that data myself to verify that everything is ok. I do not think there is any product available with the "look and feel" experience will give us.

What can we do about Distributed Denial of Service attacks?

A classic example that remains a problem is a web site under a DDoS attack in which the attacker is not yet identified (and hence the false requests can't be distinguished from the true ones). Distributed Denial of Service attacks are indeed a very special situation and ought to be rejected already at the network level. It would be a very tough requirement to size an OLTP system for the coordinated simultaneous attack of millions of botnet PC's hijacked by cybercriminals. But systems ought to be sized to handle the maximum conceivable genuine workload.

If you usually share the load between your systems, a DDoS attack will hit all the systems. So at the very beginning we already decided against such a load-sharing. In our active/active architecture, both systems have their own communication lines with different addresses (X.25 and TCP/IP). The customer *can use whatever system he chooses, but he has to be able to switch the communication to the other system. So a DDOS attack would have to deal with 2 systems.*

The ultimate is to never run out of capacity. Damian Ward of VocaLink espouses this philosophy: [2]

We run a 100% available service that uses 2 HP NonStop servers deployed in an Active/Active configuration. Our business is "risk averse," so we operate a policy where a peak day can be processed by a single NonStop system with 1 CPU down. Additionally, in this configuration, no single CPU can be operating at greater than 80% utilisation.

Imagine a 6 CPU system, (capacity 600), and remove a CPU from the theoretical model giving 5 cpu's (capacity 500). Then multiply the 5 remaining CPU's by 80% which actually only leaves us with a capacity of 400 (4 CPU's worth in the model) to do work.

The reality is that this single node actually has 6 CPU's capacity, so it already has 50% more than should be required to do the work. If we double this up to 2 sites; we have sized the system at a max capacity of 400 (4 CPU's), but in really 99.9%+ of the time we actually have a capacity of 1200 (12 CPU's) - 3 times that required to process our maximum expected workload.

Likewise we have a separate HSM pool associated with each active NonStop system. A single HSM pool (with 1 device failed) can process an entire peak day.

For telecoms we operate a policy that all of a customer's bandwidth requirement can fit down a single physical circuit at again no more than 80% utilised. However for resiliency we have 2 full size telecoms circuits to each active NonStop, (4 in total) for each customer.

So in normal running we have triple the NonStop capacity required to process the peak day which is already significantly higher than the "average" day, more than double the required HSM capacity, and quadruple the telecoms capacity required. Finally our service operates a QoS system where Synchronous payments (customer waiting 15 sec round trip SLA) always take priority over Asynchronous payments (pre-scheduled payments).In theory all synchronous payments get serviced within the SLA and 15 sec customer timeout scenario, and asynchronous ones can take a little longer to process.

So, as has been stated before, this is mostly a policy and budgets decision. What is the capacity and availability model the business is prepared to pay for in order to mitigate service risk? The availability model we use ensures significant additional capacity is always available anytime..

This may seem wasteful but is nowhere near as wasteful as a conventional production/DR model where the extra capacity is dormant and cannot easily be brought on line, especially without customer impact.

Summary

Our author, Paul Green, adds his own insights into this issue:

Great discussion; thank you, everyone. Let me relate some real-world situations that I've been involved in. Let me add that I'm specifically interested in examining this issue from the point of view of the system architect, not the end-user. I fully agree that a system that can't handle the full incoming load is less than 100% available. I also fully agree that in the real world, we must do everything possible to avoid this situation. Hats off to Damian, who works for an organization that is willing to invest the money to ensure that sufficient capacity is always available. All I can say is that most of the customers that I've dealt with over my career wouldn't dream of funding such a deluxe infrastructure.

[2] Avoiding Capacity Exhaustion, *Availability Digest*; July 2012.
http://www.availabilitydigest.com/public_articles/0707/workload_forecasting.pdf

If I divide my customers into groups based on their approach to dealing with overload situations, they fall into several broad categories:

Category #1. "Avoid the problem". This group sets a goal of always having enough capacity to handle any workload, usually has a fairly predictable workload, and usually diligently monitors their systems and takes action far enough in advance to avoid the problem. So far, so good. However, if they do get overloaded, they have no defense. Generally speaking, they run around with their hair on fire, utterly surprised and in a panic because this was never supposed to happen. Their solution is usually to rush out and buy more hardware (only to see the situation repeat in 2-3 years).

Category #2. "Bounce the application". This group has one universal solution for any application issue. Restart it. If that doesn't work, restart the operating system. Keep doing this until the problem goes away. One of my customers has a hard-coded 4-second timeout between all clients and servers. If this timeout is ever reached, code is invoked to restart the application. While the technique is brutal, it has the benefit that you can easily predict the duration of the recovery time.

Category #3. "Protect the online application, defer batch". This group has two classes of inbound transactions: those that must be processed in real time (online), and those that can take a few hours or more (batch). The batch system simply recursively uses the online system to do its work. So when the online system is at peak load, they hold off entering batch transactions. This is a priority scheme in which the end users decide which service to purchase (online or batch) based on their needs and price. As before, success generally rides on proper manual intervention.

Category #4. "Muddle through". This group muddles along with no clear strategy. They try to have enough capacity, but aren't methodical about it. They have a bunch of ad-hoc, manual techniques to use when things get rough. If they can deflect blame onto someone else, they do so. These clients are usually under-funded and under-resourced.

Category #5. "Competent and Paranoid". Very rare creature. This group always has enough hardware capacity to handle any reasonable load and most unreasonable loads. They methodically track usage and accurately predict future requirements, and they are diligent about staying ahead of the demand. Further, they obsessively run accurate benchmarks of simulated traffic so that they know how their systems (hardware and software) behave under real-world situations. They know their maximum throughput, and they know (because they've simulated it) what happens if a line to an HSM goes down, or a line to Visa goes down, or a CPU goes offline, or any of the real-world things that can go bad does go bad. They know because they injected errors in their lab. They have thought about the problem. If they have a weakness, it is that they still require too much manual intervention. But otherwise, they are the best of the bunch.

I hope we can convince IT shops to adopt a defense-in-depth approach. Buying enough hardware early and often enough is necessary but not sufficient. The rest is up to the application architects.

I rather like the idea of introducing some randomness into the basket of possible solutions to load shedding. I think there is an analogy that can be made to the packet flow control algorithms of the Internet. I'm not an expert in packet flow control, so I won't try to go into details, but as I understand it, many devices today have what is called a "tail drop" or "tail truncate" algorithm -- they capture packets until their buffers fill up, and then they drop all incoming packets. The problem with such an algorithm is that it produces bursty traffic and introduces long latencies into the transmission of data. A new algorithm has been proposed in which the receiver randomly discards packets as its queue fills up. The longer the queue, the more packets get dropped. The act of discarding packets provides a clue to the sender that he's transmitting too fast. His algorithm will slow him down. Communication continues and is never interrupted. It is fair because everyone is treated the same way. The pipe is still used efficiently (high degree of bandwidth utilization). Latency is kept low. Well-behaved senders are not penalized. It seems to me that these are some of the same qualities we'd like to see in any load-shedding algorithm.

(If you are interested in reading about the newly-proposed load-shedding algorithm for TCP/IP, do an Internet search for "Controlling Queue Delay" by Kathleen Nichols and Van Jacobson.)

I'll bet the flow-control algorithms (if any) used in POS/ATM terminals still date from the time of low-speed, dial-up lines. But now that everything is online, why not come up with a consistent, unified approach? Instead of sending the request and starting up a 60-second timer that will abort the request (which is what many devices still do), [why not] have them retry sooner but back off an increasing amount of time each time they time out? If the whole network worked this way, then when the central server(s) got saturated, the system would automatically find the appropriate input rate that would match the processing capacity of the system. If the servers were only slightly overloaded, then most requests would still get processed on the first try, but some clients, chosen at random, would take a little longer. As the servers get more overloaded, more clients are delayed.

Work smarter, not harder.

Concluding Thoughts

Clearly, dealing with overloaded application processing systems is a real problem that deserves serious and sustained effort. We believe it is important to factor these concerns into both the initial design of the system and the ongoing maintenance. While the precise methods will vary from one application to another, due to the unique properties of each application, we can still draw some important lessons from this discussion.

First, every effort must be made to design a system that cannot become overloaded under any reasonable (or even moderately unreasonable) situation. Second, all affected parties should be involved in the up-front design of any load-shedding algorithms: senior management, end-users, application designers, and system vendors. Each of these parties has a unique role and set of insights. Third, to the greatest extent possible, the design should permit the software to dynamically take care of itself. While there will always be a need for manual intervention, automated interventions can be designed and tested under a wide variety of simulated conditions; and thus an organization can be assured that many situations will be quickly and effectively managed. Indeed, advance testing of system behavior, capacity and response time is fundamental to success of continuously-available systems. Fourth, there is a strong need for continuous monitoring, logging, and in some cases, publication of key system parameters. Certainly, operations staff requires detailed knowledge of system behavior. But in addition, users can often modify their own behavior if they know the system is overloaded; at the very least, they will appreciate the disclosure of anticipated response times. (Think of the difference between a call center that informs you of your anticipated wait time and one that does not do so). Lastly, building a culture that strives to provide service under all possible circumstances ensures that an organization will succeed.

Acknowledgements

We would like to thank the following Continuous Availability Forum members for their contributions to this topic:

Jon Schmidt
Gerhard Schwartz
Thomas Burg
Robert Gezelter
Wolfgang Breidbach
Tom Wright
Damian Ward
Alan Smith
Mike Fyson
Randall Becker