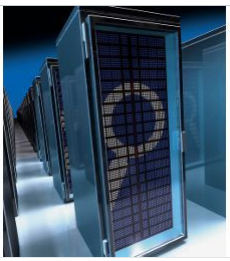
CONTRIBUTED ARTICLES

The Tail at Scale

By Jeffrey Dean, Luiz André Barroso
Communications of the ACM, February 2013, Vol. 56 No. 2, Pages 74-80
10.1145/2408776.2408794
Comments



Credit: Giacomo Marchesi

Systems that respond to user actions quickly (within 100ms) feel more fluid and natural to users than those that take longer.[3] Improvements in Internet connectivity and the rise of warehouse-scale computing systems[2] have enabled Web services that provide fluid responsiveness while consulting multi-terabyte datasets spanning thousands of servers; for example, the Google search system updates query results interactively as the user types, predicting the most likely query based on the prefix typed so far, performing the search and showing the results within a few tens of milliseconds. Emerging augmented-reality devices (such as the Google Glass prototype[7]) will need associated Web services with even greater responsiveness in order to guarantee seamless interactivity.

Back to Top

Key Insights

- Even rare performance hiccups affect a significant fraction of all requests in large-scale distributed systems.
- Eliminating all sources of latency variability in large-scale systems is impractical, especially in shared environments.
- Using an approach analogous to fault-tolerant computing, tail-tolerant software techniques form a predictable whole out of less-predictable parts.

It is challenging for service providers to keep the tail of latency distribution short for interactive services as the size and complexity of the system scales up or as overall use increases. Temporary high-latency episodes (unimportant in moderate-size systems) may come to dominate overall service performance at large scale. Just as fault-tolerant computing aims to create a reliable whole out of less-reliable parts, large online services need to create a predictably responsive whole out of less-predictable parts; we refer to such systems as "latency tail-tolerant," or simply "tail-tolerant." Here, we outline some common causes for high-latency episodes in large online services and describe techniques that reduce their severity or mitigate their effect on whole-system performance. In many cases, tail-tolerant techniques can take advantage of resources already deployed to achieve fault-tolerance, resulting in low additional overhead. We explore how these techniques allow system utilization to be driven higher without lengthening the latency tail, thus avoiding wasteful overprovisioning.

Back to Top

Why Variability Exists?

Variability of response time that leads to high tail latency in individual components of a service can arise for many reasons, including:

Shared resources. Machines might be shared by different applications contending for shared resources (such as CPU cores, processor caches, memory bandwidth, and network bandwidth), and within the same application different requests might contend for resources;

Daemons. Background daemons may use only limited resources on average but when scheduled can generate multi-millisecond hiccups;

Global resource sharing. Applications running on different machines might contend for global resources (such as network switches and shared file systems);

Maintenance activities. Background activities (such as data reconstruction in distributed file systems, periodic log compactions in storage systems like BigTable,[4] and periodic garbage collection in garbage-collected languages) can cause periodic spikes in latency; and

Queueing. Multiple layers of queueing in intermediate servers and network switches amplify this variability.

Increased variability is also due to several hardware trends:

Power limits. Modern CPUs are designed to temporarily run above their average power envelope, mitigating thermal effects by throttling if this activity is sustained for a long period;[5]

Garbage collection. Solid-state storage devices provide very fast random read access, but the need to periodically garbage collect a large number of data blocks can increase read latency by a factor of 100 with even a modest level of write activity; and

Energy management. Power-saving modes in many types of devices save considerable energy but add additional latency when moving from inactive to active modes.

Back to Top

Component-Level Variability Amplified By Scale

A common technique for reducing latency in large-scale online services is to parallelize sub-operations across many different machines, where each sub-operation is co-located with its portion of a large dataset. Parallelization happens by fanning out a request from a root to a large number of leaf servers and merging responses via a request-distribution tree. These sub-operations must all complete within a strict deadline for the service to feel responsive.

Variability in the latency distribution of individual components is magnified at the service level; for example, consider a system where each server typically responds in 10ms but with a 99th-percentile latency of one second. If a user request is

handled on just one such server, one user request in 100 will be slow (one second). The figure here outlines how service-level latency in this hypothetical scenario is affected by very modest fractions of latency outliers. If a user request must collect responses from 100 such servers in parallel, then 63% of user requests will take more than one second (marked "x" in the figure). Even for services with only one in 10,000 requests experiencing more than one-second latencies at the single-server level, a service with 2,000 such servers will see almost one in five user requests taking more than one second (marked "o" in the figure).

Table 1 lists measurements from a real Google service that is logically similar to this idealized scenario; root servers distribute a request through intermediate servers to a very large number of leaf servers. The table shows the effect of large fan-out on latency distributions. The 99th-percentile latency for a single random request to finish, measured at the root, is 10ms. However, the 99th-percentile latency for all requests to finish is 140ms, and the 99th-percentile latency for 95% of the requests finishing is 70ms, meaning that waiting for the slowest 5% of the requests to complete is responsible for half of the total 99%-percentile latency. Techniques that concentrate on these slow outliers can yield dramatic reductions in overall service performance.

Overprovisioning of resources, careful real-time engineering of software, and improved reliability can all be used at all levels and in all components to reduce the base causes of variability. We next describe general approaches useful for reducing variability in service responsiveness.

Back to Top

Reducing Component Variability

Interactive response-time variability can be reduced by ensuring interactive requests are serviced in a timely manner through many small engineering decisions, including:

Differentiating service classes and higher-level queuing. Differentiated service classes can be used to prefer scheduling requests for which a user is waiting over non-interactive requests. Keep low-level queues short so higher-level policies take effect more quickly; for example, the storage servers in Google's cluster-level file-system software keep few operations outstanding in the operating system's disk queue, instead maintaining their own priority queues of pending disk requests. This shallow queue allows the servers to issue incoming high-priority interactive requests before older requests for latency-insensitive batch operations are served.

Reducing head-of-line blocking. High-level services can handle requests with widely varying intrinsic costs. It is sometimes useful for the system to break long-running requests into a sequence of smaller requests to allow interleaving of the execution of other short-running requests; for example, Google's Web search system uses such time-slicing to prevent a small number of very computationally expensive queries from adding substantial latency to a large number of concurrent cheaper queries.

Managing background activities and synchronized disruption. Background tasks can create significant CPU, disk, or network load; examples are log compaction in log-oriented storage systems and garbage-collector activity in garbage-collected languages. A combination of throttling, breaking down heavyweight operations into smaller operations, and triggering such operations at times of lower overall load is often able to reduce the effect of background activities on interactive request latency. For large fan-out services, it is sometimes useful for the system to synchronize the background activity across many different machines. This synchronization enforces a brief burst of activity on each machine simultaneously, slowing only those interactive requests being handled during the brief period of background activity. In contrast, without synchronization, a few machines are always doing some background activity, pushing out the latency tail on all requests.

Missing in this discussion so far is any reference to caching. While effective caching layers can be useful, even a necessity in some systems, they do not directly address tail latency, aside from configurations where it is guaranteed that the entire working set of an application can reside in a cache.

Back to Top

Living with Latency Variability

The careful engineering techniques in the preceding section are essential for building high-performance interactive services, but the scale and complexity of modern Web services make it infeasible to eliminate all latency variability. Even if such perfect behavior could be achieved in isolated environments, systems with shared computational resources exhibit performance fluctuations beyond the control of application developers. Google has therefore found it advantageous to develop tail-tolerant techniques that mask or work around temporary latency pathologies, instead of trying to eliminate them altogether. We separate these techniques into two main classes: The first corresponds to within-request immediate-response techniques that operate at a time scale of tens of milliseconds, before longer-term techniques have a chance to react. The second consists of cross-request long-term adaptations that perform on a time scale of tens of seconds to minutes and are meant to mask the effect of longer-term phenomena.

Back to Top

Within Request Short-Term Adaptations

A broad class of Web services deploy multiple replicas of data items to provide additional throughput capacity and maintain availability in the presence of failures. This approach is particularly effective when most requests operate on largely read-only, loosely consistent datasets; an example is a spelling-correction service that has its model updated once a day while handling thousands of correction requests per second. Similarly, distributed file systems may have multiple replicas of a given data chunk that can all be used to service read requests. The techniques here show how replication can also be used to reduce latency variability within a single higher-level request:

Hedged requests. A simple way to curb latency variability is to issue the same request to multiple replicas and use the results from whichever replica responds first. We term such requests "hedged requests" because a client first sends one request to the replica believed to be the most appropriate, but then falls back on sending a secondary request after some brief delay. The client cancels remaining outstanding requests once the first result is received. Although naive implementations of this technique typically add unacceptable additional load, many variations exist that give most of the latency-reduction effects while increasing load only modestly.

One such approach is to defer sending a secondary request until the first request has been outstanding for more than the 95th-percentile expected latency for this class of requests. This approach limits the additional load to approximately 5% while substantially shortening the latency tail. The technique works because the source of latency is often not inherent in the particular request but rather due to other forms of interference. For example, in a Google benchmark that reads the values for 1,000 keys stored in a BigTable table distributed across 100 different servers, sending a hedging request after a 10ms delay reduces the 99.9th-percentile latency for retrieving all 1,000 values from 1,800ms to 74ms while sending just 2% more requests. The overhead of hedged requests can be further reduced by tagging them as lower priority than the primary requests.

Tied requests. The hedged-requests technique also has a window of vulnerability in which multiple servers can execute the same request unnecessarily. That extra work can be capped by waiting for the 95th-percentile expected latency before issuing the hedged request, but this approach limits the benefits to only a small fraction of requests. Permitting more aggressive use of hedged requests with moderate resource consumption requires faster cancellation of requests.

A common source of variability is queueing delays on the server before a request begins execution. For many services, once a request is actually scheduled and begins execution, the variability of its completion time goes down substantially. Mitzenmacher[10] said allowing a client to choose between two servers based on queue lengths at enqueue time exponentially improves load-balancing performance over a uniform random scheme. We advocate not choosing but rather enqueuing copies of a request in multiple servers simultaneously and allowing the servers to communicate updates on the status of these copies to each other. We call requests where servers perform cross-server status updates "tied

requests." The simplest form of a tied request has the client send the request to two different servers, each tagged with the identity of the other server ("tied"). When a request begins execution, it sends a cancellation message to its counterpart. The corresponding request, if still enqueued in the other server, can be aborted immediately or deprioritized substantially.

There is a brief window of one average network message delay where both servers may start executing the request while the cancellation messages are both in flight to the other server. A common case where this situation can occur is if both server queues are completely empty. It is useful therefore for the client to introduce a small delay of two times the average network message delay (1ms or less in modern data-center networks) between sending the first request and sending the second request.

Google's implementation of this technique in the context of its cluster-level distributed file system is effective at reducing both median and tail latencies. Table 2 lists the times for servicing a small read request from a BigTable where the data is not cached in memory but must be read from the underlying file system; each file chunk has three replicas on distinct machines. The table includes read latencies observed with and without tied requests for two scenarios: The first is a cluster in which the benchmark is running in isolation, in which case latency variability is mostly from self-interference and regular cluster-management activities. In it, sending a tied request that does cross-server cancellation to another file system replica following 1ms reduces median latency by 16% and is increasingly effective along the tail of the latency distribution, achieving nearly 40% reduction at the 99.9^{th}-percentile latency. The second scenario is like the first except there is also a large, concurrent sorting job running on the same cluster contending for the same disk resources in the shared file system. Although overall latencies are somewhat higher due to higher utilization, similar reductions in the latency profile are achieved with the tied-request technique discussed earlier. The latency profile with tied requests while running a concurrent large sorting job is nearly identical to the latency profile of a mostly idle cluster without tied requests. Tied requests allow the workloads to be consolidated into a single cluster, resulting in dramatic computing cost reductions. In both Table 2 scenarios, the overhead of tied requests in disk utilization is less than 1%, indicating the cancellation strategy is effective at eliminating redundant reads.

An alternative to the tied-request and hedged-request schemes is to probe remote queues first, then submit the request to the least-loaded server.[10] It can be beneficial but is less effective than submitting work to two queues simultaneously for three main reasons: load levels can change between probe and request time; request service times can be difficult to estimate due to underlying system and hardware variability; and clients can create temporary hot spots by all clients picking the same (least-loaded) server at the same time. The Distributed Shortest-Positioning Time First system[9] uses another variation in which the request is sent to one server and forwarded to replicas only if the initial server does not have it in its cache and uses cross-server cancellations.

Worth noting is this technique is not restricted to simple replication but is also applicable in more-complex coding schemes (such as Reed-Solomon) where a primary request is sent to the machine with the desired data block, and, if no response is received following a brief delay, a collection of requests is issued to a subset of the remaining replication group sufficient to reconstruct the desired data, with the whole ensemble forming a set of tied requests.

Note, too, the class of techniques described here is effective only when the phenomena that causes variability does not tend to simultaneously affect multiple request replicas. We expect such uncorrelated phenomena are rather common in large-scale systems.

Back to Top

Cross-Request Long-Term Adaptations

Here, we turn to techniques that are applicable for reducing latency variability caused by coarser-grain phenomena (such as service-time variations and load imbalance). Although many systems try to partition data in such a way that the partitions have equal cost, a static assignment of a single partition to each machine is rarely sufficient in practice for two reasons: First, the performance of the underlying machines is neither uniform nor constant over time, for reasons (such as thermal throttling and shared workload interference) mentioned earlier. And second, outliers in the assignment of items to partitions can cause data-induced load imbalance (such as when a particular item becomes popular and the load for its partition increases).

Micro-partitions. To combat imbalance, many of Google's systems generate many more partitions than there are machines in the service, then do dynamic assignment and load balancing of these partitions to particular machines. Load balancing is then a matter of moving responsibility for one of these small partitions from one machine to another. With an average of, say, 20 partitions per machine, the system can shed load in roughly 5% increments and in $1/20^{th}$ the time it would take if the system simply had a one-to-one mapping of partitions to machines. The BigTable distributed-storage system stores data in tablets, with each machine managing between 20 and 1,000 tablets at a time. Failure-recovery speed is also improved through micro-partitioning, since many machines pick up one unit of work when a machine failure occurs. This method of using micro-partitions is similar to the virtual servers notion as described in Stoica[12] and the virtual-processor-partitioning technique in DeWitt et al.[6]

Selective replication. An enhancement of the micro-partitioning scheme is to detect or even predict certain items that are likely to cause load imbalance and create additional replicas of these items. Load-balancing systems can then use the additional replicas to spread the load of these hot micro-partitions across multiple machines without having to actually move micro-partitions. Google's main Web search system uses this approach, making additional copies of popular and important documents in multiple micro-partitions. At various times in Google's Web search system's evolution, it has also created micro-partitions biased toward particular document languages and adjusted replication of these micro-partitions as the mix of query languages changes through the course of a typical day. Query mixes can also change abruptly, as when, say, an Asian data-center outage causes a large fraction of Asian-language queries to be directed to a North American facility, materially changing its workload behavior.

Latency-induced probation. By observing the latency distribution of responses from the various machines in the system, intermediate servers sometimes detect situations where the system performs better by excluding a particularly slow machine, or putting it on probation. The source of the slowness is frequently temporary phenomena like interference from unrelated networking traffic or a spike in CPU activity for another job on the machine, and the slowness tends to be noticed when the system is under greater load. However, the system continues to issue shadow requests to these excluded servers, collecting statistics on their latency so they can be reincorporated into the service when the problem abates. This situation is somewhat peculiar, as removal of serving capacity from a live system during periods of high load actually improves latency.

Back to Top

Large Information Retrieval Systems

In large information-retrieval (IR) systems, speed is more than a performance metric; it is a key quality metric, as returning good results quickly is better than returning the best results slowly. Two techniques apply to such systems, as well as other to systems that inherently deal with imprecise results:

Good enough. In large IR systems, once a sufficient fraction of all the leaf servers has responded, the user may be best served by being given slightly incomplete ("good-enough") results in exchange for better end-to-end latency. The chance that a particular leaf server has the best result for the query is less than one in 1,000 queries, odds further reduced by replicating the most important documents in the corpus into multiple leaf servers. Since waiting for exceedingly slow servers might stretch service latency to unacceptable levels, Google's IR systems are tuned to occasionally respond with good-enough results when an acceptable fraction of the overall corpus has been searched, while being careful to ensure good-enough results remain rare. In general, good-enough schemes are also used to skip nonessential subsystems to improve responsiveness; for example, results from ads or spelling-correction systems are easily skipped for Web searches if they do not respond in time.

A simple way to curb latency variability is to issue the same request to multiple replicas and use the results from whichever replica responds first.

Canary requests. Another problem that can occur in systems with very high fan-out is that a particular request exercises an untested code path, causing crashes or extremely long delays on thousands of servers simultaneously. To prevent such correlated crash scenarios, some of Google's IR systems employ a technique called "canary requests"; rather than initially send a request to thousands of leaf servers, a root server sends it first to one or two leaf servers. The remaining servers are only queried if the root gets a successful response from the canary in a reasonable period of time. If the server crashes or hangs while the canary request is outstanding, the system flags the request as potentially dangerous and prevents further execution by not sending it to the remaining leaf servers. Canary requests provide a measure of robustness to back-ends in the face of difficult-to-predict programming errors, as well as malicious denial-of-service attacks.

The canary-request phase adds only a small amount of overall latency because the system must wait for only a single server to respond, producing much less variability than if it had to wait for all servers to respond for large fan-out requests; compare the first and last rows in Table 1. Despite the slight increase in latency caused by canary requests, such requests tend to be used for every request in all of Google's large fan-out search systems due to the additional safety they provide.

Back to Top

Mutations

The techniques we have discussed so far are most applicable for operations that do not perform critical mutations of the system's state, which covers a broad range of data-intensive services. Tolerating latency variability for operations that mutate state is somewhat easier for a number of reasons: First, the scale of latency-critical modifications in these services is generally small. Second, updates can often be performed off the critical path, after responding to the user. Third, many services can be structured to tolerate inconsistent update models for (inherently more latency-tolerant) mutations. And, finally, for those services that require consistent updates, the most commonly used techniques are quorum-based algorithms (such as Lamport's Paxos[8]); since these algorithms must commit to only three to five replicas, they are inherently tail-tolerant.

Back to Top

Hardware Trends and Their Effects

Variability at the hardware level is likely to be higher in the future due to more aggressive power optimizations becoming available and fabrication challenges at deep submicron levels resulting in device-level heterogeneity. Device heterogeneity combined with ever-increasing system scale will make tolerating variability through software techniques even more important over time. Fortunately, several emerging hardware trends will increase the effectiveness of latency-tolerating techniques. For example, higher bisection bandwidth in data-center networks and network-interface optimizations that reduce per-message overheads (such as remote direct-memory access) will reduce the cost of tied requests, making it more likely that cancellation messages are received in time to avoid redundant work. Lower per-message overheads naturally allow more fine-grain requests, contributing to better multiplexing and avoiding head-of-line blocking effects.

Back to Top

Conclusion

Delivering the next generation of compute-intensive, seamlessly interactive cloud services requires consistently responsive massive-scale computing systems that are only now beginning to be contemplated. As systems scale up, simply stamping out all sources of performance variability will not achieve such responsiveness. Fault-tolerant techniques were developed because guaranteeing fault-free operation became infeasible beyond certain levels of system complexity. Similarly, tail-tolerant techniques are being developed for large-scale services because eliminating all sources of variability is also infeasible. Although approaches that address particular sources of latency variability are useful, the most powerful tail-tolerant techniques reduce latency hiccups regardless of root cause. These tail-tolerant techniques allow designers to continue to optimize for the common case while providing resilience against uncommon cases. We have outlined a small collection of tail-tolerant techniques that have been effective in several of Google's large-scale software systems. Their importance will only increase as Internet services demand ever-larger and more complex warehouse-scale systems and as the underlying hardware components display greater performance variability.

While some of the most powerful tail-tolerant techniques require additional resources, their overhead can be rather modest, often relying on existing capacity already provisioned for fault-tolerance while yielding substantial latency improvements. In addition, many of these techniques can be encapsulated within baseline libraries and systems, and the latency improvements often enable radically simpler application-level designs. Besides enabling low latency at large scale, these techniques make it possible to achieve higher system utilization without sacrificing service responsiveness.

Back to Top

Acknowledgments

We thank Ben Appleton, Zhifeng Chen, Greg Ganger, Sanjay Ghemawat, Ali Ghodsi, Rama Govindaraju, Lawrence Greenfield, Steve Gribble, Brian Gustafson, Nevin Heintze, Jeff Mogul, Andrew Moore, Rob Pike, Sean Quinlan, Gautham Thambidorai, Ion Stoica, Amin Vahdat, and T.N. Vijaykumar for their helpful feedback on earlier drafts and presentations of this work. Numerous people at Google have worked on systems that use these techniques.

Back to Top

References

1. Barroso, L.A. and Höelzle, U. The case for energy proportional computing. *IEEE Computer 40*, 12 (Dec. 2007), 3337.

2. Barroso, L.A. and Höelzle, U. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-scale Machines*. Synthesis Series on Computer Architecture, Morgan & Claypool Publishers, May 2009.

3. Card, S.K., Robertson, G.G., and Mackinlay, J.D. The information visualizer: An information workspace. In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems* (New Orleans, Apr. 28May 2). ACM Press, New York, 1991, 181188.

4. Chang F., Dean J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R.E. BigTable: A distributed storage system for structured data. In *Proceedings of the Seventh Symposium on Operating Systems Design and Implementation* (Seattle, Nov.). USENIX Association, Berkeley CA, 2006, 205218.

5. Charles, J., Jassi, P., Ananth, N.S., Sadat, A., and Fedorova, A. Evaluation of the Intel Core i7 Turbo Boost feature. In *Proceedings of the IEEE International Symposium on Workload Characterization* (Austin, TX, Oct. 46). IEEE Computer Society Press, 2009, 188197.

6. DeWitt, D.J., Naughton, J.F., Schneider, D.A., and Seshadri, S. Practical skew handling in parallel joins. In *Proceedings of the 18th International Conference on Very Large Data Bases*, Li-Yan Yuan, Ed. (Vancouver, BC, Aug. 2427). Morgan Kaufmann Publishers, Inc., San Francisco, 1992, 2740.

7. Google, Inc. Project Glass; http://g.co/projectglass

8. Lamport, L. The part-time parliament. *ACM Transactions on Computer Systems 16*, 2 (May 1998), 133169.

9. Lumb, C.R. and Golding, R. D-SPTF: Decentralized request distribution in brick-based storage systems. *SIGOPS Operating System Review 38*, 5 (Oct. 2004), 3747.

10. Mitzenmacher, M. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Computing 12*, 10 (Oct. 2001), 10941104.

11. Mudge, T. and Hölzle, U. Challenges and opportunities for extremely energy-efficient processors. *IEEE Micro 30*, 4 (July 2010), 2024.

12. Stoica I., Morris, R., Karger, D., Kaashoek, F., and Balakrishnan, H. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proceedings of SIGCOMM* (San Diego, Aug. 2731). ACM Press, New York, 2001, 149160.

Back to Top

Authors

Jeffrey Dean (jeff@google.com) is a Google Fellow in the Systems Infrastructure Group of Google Inc., Mountain View, CA.

Luiz André Barroso (luiz@google.com) is a Google Fellow and technical lead of core computing infrastructure at Google Inc., Mountain View, CA.

Back to Top

Figures

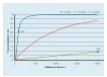


Figure. Probability of one-second service-level response time as the system scales and frequency of server-level high-latency outliers varies.

Back to Top

Tables

Table 1. Individual-leaf-request finishing times for a large fan-out service tree (measured from root node of the tree).

Table 2. Read latencies observed in a BigTable service benchmark.

Back to top

No entries found