Twitter Streams: When Scale Disrupts Flow

David Gary
Department of Computing and Informatics
The University of North Carolina at Charlotte
Charlotte, NC, USA
dgary9@uncc.edu

***Abstract*—Scaling any kind of software system is notoriously difficult. As new requirements appear, evaluating the added complexity of proposed solutions can be challenging as the principles of proper system design are not consistent at different scales. This has shown true through two decades of major tech companies expanding and exploring distributed system architecture in larger and larger sizes. In this paper, we aim to break down the evolution of Twitter's data stream architecture, with the goal of identifying complexity contribution to ultimately gain insight on common pitfalls in scaling design. Through visualizations, transformations that reflect metric-goal shifts, and the use of data-driven techniques, we explicitly show the flaws in what seem to be sensible system architecture choices.**

***Index Terms*—Distributed Systems, Scaling System Architecture, Twitter**

I. Introduction

As technology companies expanded throughout the 2000s and into today, principles of system design finally faced real world tests. Decades of theoretical research based on mathematical principles were finally put into test, and the industry was forced to find areas where limitations were hiding. Doing so is immensely challenging, especially under the pressure of achieving market demand in time. Within the notable tech companies that attempted to scale in response to rapid growth, different philosophical approaches emerged as well.

These approaches can be simplified down into two broad categories: theory-first and quick-reactionary, the latter of which Twitter [1] and Facebook [2] perfectly exemplify. Whenever the servicing scale limits for either company were being pushed, they would design a patch to their system and publish a reputable paper about it. This trial and error method worked great, but there are clearly errors to this type of approach on a principle level.

Jeff Dean has been famously quoted for saying that systems should only be planned out to satisfy 10-30x of their current demand [3]. But this advice is not firmly rooted in some mathematical law that relates to distributed system design. It was more meant to guide people through unknown territory and serve as a reference for how far in advance we are actually capable of planning when dealing with new system design territory. In fairness, this proved to be solid advice, as Google was easily the most successful company to deploy effective and massive distributed systems. Still though, the efficacy of Dean's advice does not provide a clear roadmap for why systems break at certain scales, and there seems to be a shocking lack in the current body of research on this topic. A few papers have successfully done this for matters related to distributed system design, like the works on TCP Incast Collapse [4], but not much other work exists, even to this day. This is why the market-driven developments should be explicit in how they identified and solved the problems they faced at new scales.

Unfortunately, though, Twitter's patch papers rarely addressed the underlying issues as to why larger-scales broke their architecture. To make matters even more confusing, Twitter would often utilize different metrics across each patch release to evaluate the current performance against previous, problematic designs. This only obfuscates the efficacy of a system, and their diagrams did not clarify any of these issues either. They were certainly not the only company to do so. Facebook's explanation for how they scaled a distributed hash table called Memcache is somewhat notorious for hiding and excusing the design mistakes they made [2].

Now that the collective group of large tech companies have explored designing systems at an enormous scale, there is a chance to look back and identify the problematic system components that broke down at larger scales. This paper aims to do just that. For the sake of simplicity and respecting assignment length restrictions, this analysis will be limited to the Storm [5], Heron [6], and Kafka [1] systems. The content of these three introduction papers varies in focus and explanation, leaving only a few points to narrow in on. It should be noted that some consistently developing features in each system (like semantic atomicity) are discussed in the papers but not analyzed in this paper.

By establishing clear upper and lower bounds on system performance metrics, interpolating values listed across different reports, and generating easily understandable visualizations, we analyze the development of Twitter's data stream architecture. This analysis is then used to point out components that we believe to be the root of problems faced as their systems scaled. In addition to this causal analysis, visualizations of system architectures will also be provided in the hopes that they are more informative than the original ones by Twitter.

II. State of the Art Work

This section is broken down into two parts: System Design Principles II-A and Twitter's System History II-B. The system design principles subsection reviews the principles of system

design in a broad sense and then gives a brief overview of the design approach used in a different system's history (Facebook's use of Memcache). Finally, we look at Twitter's development history, using figures from the original papers to visualize the components at each stage.

A. System Design Principles

Long before the executives at Twitter were writing code, Butler Lampson published a paper on the principles of system design [7]. Although this advice paper came out in the early days of distributed systems (circa 1983), a shocking amount of this advice stays relevant in the current landscape of distributed systems. Lampson mentions the importance of evaluating speed on an end-to-end basis across the whole system, not just the performance of a single component. He continues to say that there is a need for testing performance of each component individually, but that by prioritizing end-to-end speed, unplanned complications and connective components are accounted for. Some tips he provides run counter to what Twitter starts with, like his advice to exclusively use atomic actions. Twitter's exploration of how exactly-once and at-least semantics impact performance initially violated this, but later developments provided corrective structures. But above all, two of his tips are the most important for analyzing Twitter's development: (1) Keep it simple, stupid and (2) Utilize batch processing.

Facebook had to deal with similarly complicated data flow problems as it expanded it's architecture to meet scale, but they did not publish as many papers about each step in the process. Instead, they released a paper in 2013 [2] that described their total evolution of the system architecture to that point. Publishing all developments at once meant that the performance metrics and diagrams stayed consistent throughout the explanation. Where this work faltered came from the justifications provided for design decisions. They refused to use anything other than PHP as their base language, which ultimately meant that when language performance was critical, they had to engineer an entirely new compiler to translate PHP to C++. That choice to stick with the broken persisted in their design, as they would rarely change the function of a component. New parts were added to the system to account for whatever failure-at-scale they encountered. Facebook's success was not visibly hindered by this patchwork design pattern, showing that there is some merit to their approach.

B. Twitter's System History

This paper's review begins with Twitter's launch of Storm [5], a distributed system that provided real-time stream data processing, which is a core component to Twitter usage. Storm was presented as being highly resilient to failures at increased scale, but it's flaws were addressed from the beginning. At the core, most operations performed by Storm were simple, as shown in Fig. 1. The system's general architecture maintained simplicity through its pyramid-like structure, with the top level being the primary coordinator Nimbus. In the middle sections of the hierarchy, the components (Zookeeper and Supervisors) are meant to coordinate which Workers (the bottom of the chain) tasks will be sent to. All components have a two-way line of communication with each other, limited to heartbeat messages informing other components that they were still alive and working. The architects of Storm beleived this would allow for outages to be easily detected and handled. See Fig. 2 for higher-level view of the system's architecture.

Twitter describes the interconnection of the different system components as a network of Spouts (which send data streams or coordination operations) and Bolts (which perform transformations on the data streams or coordination operations). Storm relied heavily on threading these roles within a single host machine. This meant that one node of its distributed network might be acting as multiple roles simultaneously. The hierarchical nature of Storm's design meant that when a single host machine got overloaded, the pyramidal integrity was compromised and correction operations flooded higher levels of the system. Ultimately, that would lead to immense workloads for Nimbus, revealing it to be a single point of failure in the system.

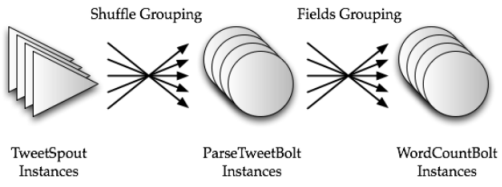

Fig. 1. Storm Worker Execution Diagram [5]

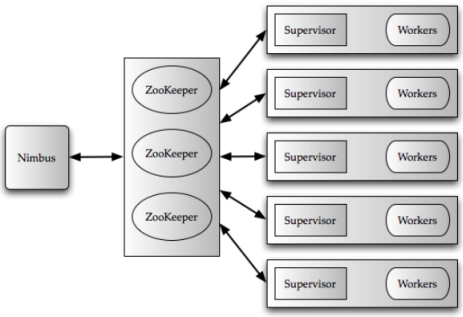

Fig. 2. Higher View Look at Twitter's Storm System [5]

Twitter then introduced Heron [6], a new architecture that accounted for the single-point of failure and interdependency problems presented in Storm. In this paper, latency diagrams were provided to show how Storm would have intense spikes in performance. An example of these diagrams is shown in Fig. 3. Heron's architecture (fully shown below in Fig. 4) was built from core components of Storm, but the hierarchical nature was redesigned to eliminate bottlenecks and single points of failure. Nimbus was eliminated, and a new top-level component, Aurora, was introduced to deploy Zookeeper topologies that would manage the flow of data and coordination operations. Some of these topologies were designated as

immediate backups, to ensure that none of the Nimbus failures came into play.

In addition, Heron introduced the concept of backpressure to the system, which expanded the heartbeat communication line and added a Metrics Manager that kept tabs on all system components. The Metrics Manager would then use the information it gathered to tell Zookeeper topologies where they could send more tasks. This paper also introduced the Heron Instance component, which was meant to cut back on the amount of host threading and interdependency. Originally, not much was explicitly told about the cross-component dependencies in Storm, but the introduction of Heron Instances revealed this in much greater depth. Heron Instances are what perform individual tasks, and they isolate task-dependency into a single, easily restartable, threaded, contained process. All in all, this was a well-designed architecture, but limitations still remained at a hardware level.

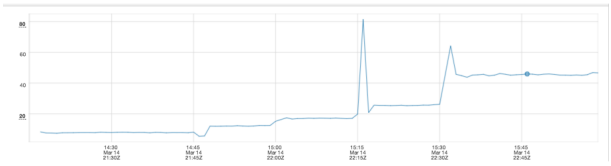

Fig. 3. Latency Diagrams from the Storm Introduction Paper [5]

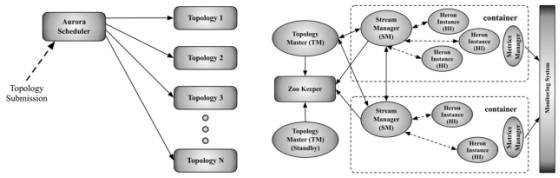

Fig. 4. Higher View Architecture of Heron [6]

The problem Twitter then faced was their lack specificity in the data stream. Many tasks were being performed in real-time that simply did not need to be, causing an immense overhead across all parts of the system. The backpressure design from Heron's architecture prevented serious outages from occurring, but overall performance was not optimal as their scale expanded. To handle this, they then implemented their Kafka architecture (a diagram of which is shown below in Fig. 5) to support delineation of task types. Instead of executing all jobs in real-time Twitter could now take advantage of batch-processing to greatly reduce the amount of necessary operations at any given time.

This made an enormously positive difference in performance, but the metrics used in this introduction paper differed from the ones used in Heron's. Both developments were comparing themselves against their predecessor, yet the metrics were not consistent. In the case of Heron, there is a slight excuse in that they adopted the policy recommended by Lampson of measuring speed on an end-to-end basis. And while that view persists in the Kafka introduction, metrics for CPU usage, latency, and throughput use inconsistent units, obfuscating the performance evolution.

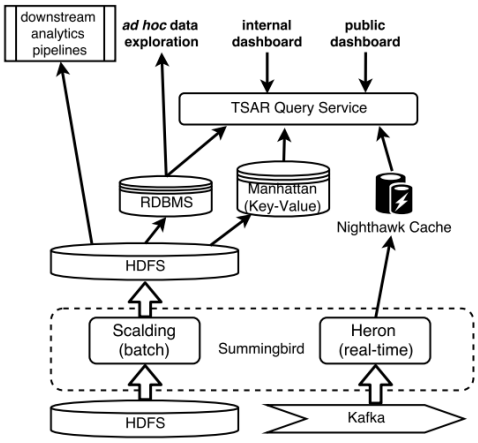

Fig. 5. Twitter's Kafka-Implementation Architecture [1]

III. Working Examples

The architectures of all three systems are often underrepresented by the figures provided in their respective papers. In light of that, diagrams have been created that mimic the original design style from each introduction but convey more information about the system's complexity and design flaws. These diagrams are exclusively representing the Storm and Heron architectures, as there is no evolution beyond Kafka to give insight on its problematic components.

The first figure (Fig. 6) shows the the structure a single machine (the Host) in the Storm system. The black arrow-lines show how the device's resources are threaded, with no returning information being passed on the single-system level. Remember that failure of one task (any node labeled T) would cause the entire system to fail.

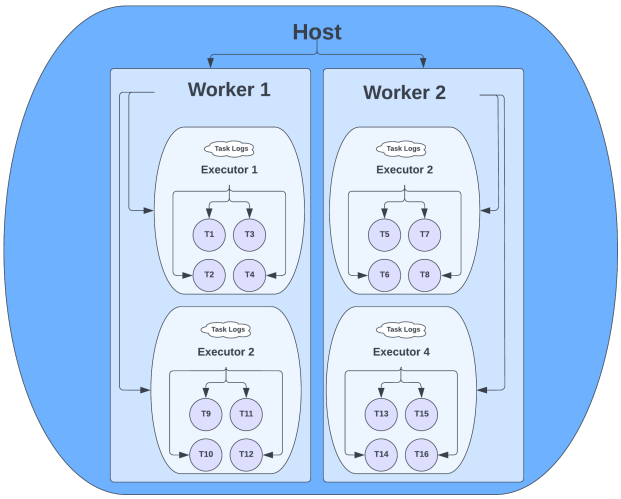

Fig. 6. Improved Storm Architecture

In Fig. 7, Aurora's delegation of topologies is shown. Backup redundancies are shown to show how this system corrected the previously existing bottleneck issues seen in Storm.

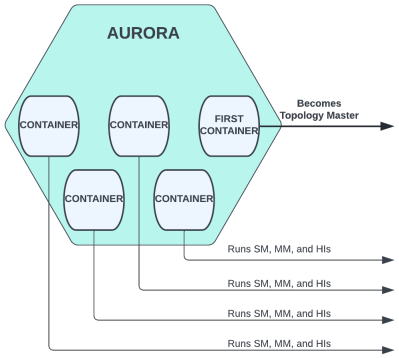

Fig. 7. Aurora Topology Setup

Then, in Fig. 8, the flow of data across Spouts and Bolts is shown. Part of the problem with previous figures is that there was little shown about the interconnectedness of data streams in Storm. That has been corrected here.

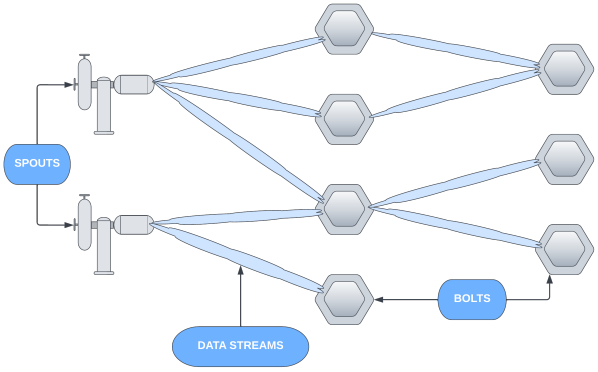

Fig. 8. Heron Data Flow

Fig. 9 shows the design of a single Heron Instance, which contains threaded dependencies to a single component of a host within the system. Stream Managers are able to track when these tasks are executed correctly because of the two-way communication maintained with the Heron Instance. Consider the difference in threaded interdepencies between the Heron architecture and the Storm architecture.

IV. Results

The visualizations listed in the previous section give better insight on the problematic components at various design levels, but none of these show how performance improved over time. As mentioned before, this is difficult to evaluate, as there are serious inconsistencies in the metrics used across the introduction papers. To handle this, a two-pronged approach was

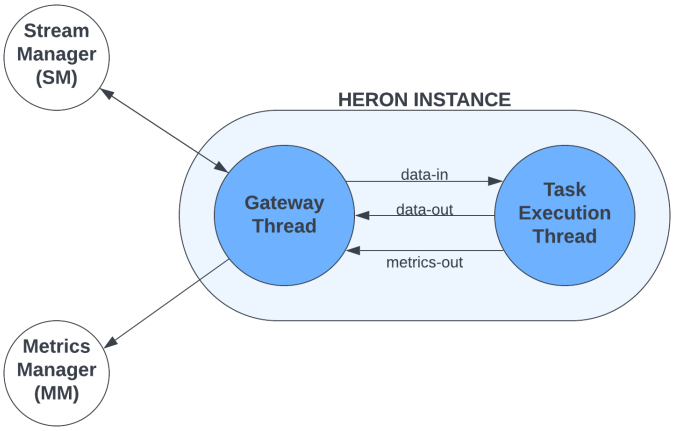

Fig. 9. Heron Instance Diagram

implemented to evaluate an overall, comparative performance metric that is visualized in Fig. 10.

The first and simplest part of this approach was to find where a magnitudinal performance difference (which we call the throughput-per-latent-second metric) was mentioned in the Heron and Kafka introductions. By chaining these together, we can say that Heron's performance ratio is 3x that of Storm and Kafka's is 20x Heron's, meaning Kafka is about 60x faster than Storm. Verifying this claim across other metrics mentioned in the papers was trickier. For that side of the overall approach, the switch to end-to-end throughput and latency measurement needed to be accounted for. By taking the Storm metrics listed in the Heron paper during comparison, we found that separated, independent component latencies needed to be summed first. With this summed latency, we then found that failure rates of mid-level components and their time taken for reshuffling and restarting needed to be converted to our throughput-per-latent-second metric's single second time scale. Through this process, we found that the throughput-per-latent-second metric was consistent with the magnitudinal differences given once an average had been made from metrics reported with and without acknowledgements being enabled in Heron. Since the values we were aiming to gather were then stoichiometrically similar to how Kafka reported its measurements, no futher conversions were deemed necessary. It should be noted that CPU usage metrics were dropped in favor of total throughput-per-latent-second metric. But other record difference subtleties like the amount of operations delegated for batch processing were accounted for.

V. Conclusion

The refinement and expansion methods used for Twitter's data stream processing system contain valuable information on how scaling up impacts the performance of key pieces, but the works that introduced these developments were often lacking in their detail. Redesigning the figures from previous system versions to convey the lessons learned is an easy path

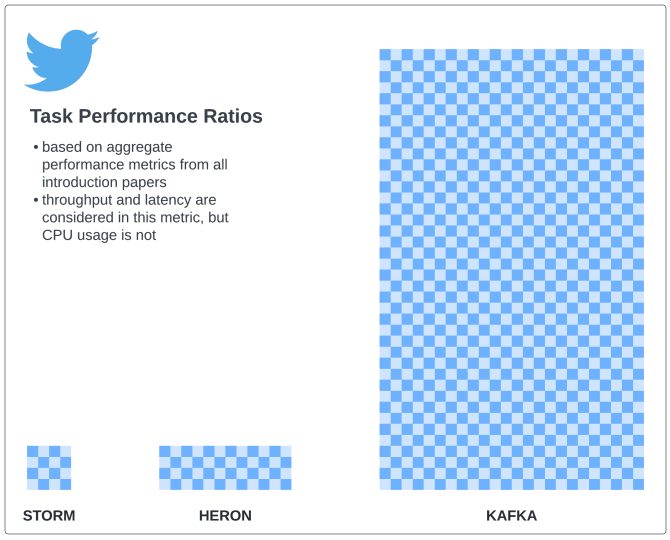

Fig. 10. Throughput Difference Across Architectures

to effective visual communication. But this step requires the ability to identify which components were problematic and why, which all of this depends on consistency of metrics and evaluation methodologies. Planning these out can be incredibly hard and might be limiting for the overall design of a system. Still, finding commonalities for comparing across developmental is likely possible and should be included more regularly in Twitter's system update releases. By pursuing those conversions and redesigning figures with the known insights, this analysis has thoroughly shown key features that led to Twitter's failures at unseen scales.

References

[1] L. Zhang and D. Malife. (2021) Processing billions of events in real time at twitter. Online. [Online]. Available: https://blog.twitter.com/engineering/en_us/topics/infrastructure/2021/processing-billions-of-events-in-real-time-at-twitter-

[2] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani, "Scaling memcache at facebook," in *Proceedings of the 2013 USENIX Symposium on Networked Systems Design and Implementation*. USENIX, 2013.

[3] J. Dean. (2010) Software engineering advice from building large-scale distributed systems. Online. [Online]. Available: https://www.cs.cornell.edu/projects/ladis2009/talks/dean-keynote-ladis2009.pdf

[4] A. Phanishayee, E. Krevat, V. Vasudevan, D. G. Andersen, G. R. Ganger, G. A. Gibson, and S. Seshan, "Measurement and analysis of tcp throughput collapse in cluster-based storage systems," in *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, ser. FAST'08. USA: USENIX Association, 2008.

[5] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy, "Storm@twitter," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 147–156. [Online]. Available: https://doi.org/10.1145/2588555.2595641

[6] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, "Twitter heron: Stream processing at scale," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 239–250. [Online]. Available: https://doi.org/10.1145/2723372.2742788

[7] B. W. Lampson, "Hints for computer system design," *SIGOPS Oper. Syst. Rev.*, vol. 17, no. 5, p. 33–48, oct 1983. [Online]. Available: https://doi.org/10.1145/773379.806614