Proximal Consensus: Responsive Data Pipelines with Approximate Fault Detection

Roy Shadmon
University of California, Santa Cruz
rshadmon@ucsc.edu

Abhishek Singh
University of California, Irvine
abhishas@uci.edu

Faisal Nawab
University of California, Irvine
nawabf@uci.edu

Owen Arden
University of California, Santa Cruz
owen@soe.ucsc.edu

Abstract

Emerging streaming Internet of Things (IoT) applications require real-time response through edge computing providers. However, these edge providers are outside of the trust domain of applications and may act maliciously. Existing methods to reach agreement across untrusted parties are infeasible for real-time applications. To this end, we propose the notion of *proximal consensus*, where nodes agree on the state of batched streaming computations even if they are not identical. Specifically, proximal consensus uses the (partial) computation of replicas and analyzes the statistical properties of these results to find an outcome in real-time and with dynamically estimated confidence levels. Central to this work is tackling the complications arising from tolerating the presence of network conditions and Byzantine failures. We explore the benefits of proximal consensus in Centauri, a framework for responsive, fault-tolerant data pipelines.

Keywords: distributed systems, consensus, fault tolerance, streaming, data pipelines

1 Introduction

Many large-scale distributed systems receive inputs from and send results to devices at the edge. At the edge, network reliability and node security is far more variable than in the data center, especially for system components with limited resources and connectivity such as sensor networks and IoT devices. One of the primary challenges to building reliable distributed applications in these scenarios is the conflict between fault tolerance and responsiveness. Replicating components helps systems tolerate a bounded number of faults, but ensuring replicas remain consistent often comes at the price of responsiveness due to the additional communication required by consensus protocols.

Systems designers frequently decide that responsiveness is more important than consistency, and settle for weak guarantees such as eventual consistency [1, 3, 6, 8, 12, 18, 20, 23]. These eventually-consistent systems often focus on the problem of data storage and indexing, but do not provide solutions for higher-level processing abstractions that are needed for streaming-based computations across edge nodes. More importantly, eventual consistency is inappropriate for security-critical or safety-critical applications where even temporarily inconsistent data could significantly compromise the functionality of the system.

For example, consider the route-planning application for emergency responders illustrated in Figure 1 that continually updates the fastest route to a target destination based on real-time traffic data. Timely, accurate route recommendations are critical, so the application replicates its components to tolerate faults and ensure availability.

When traffic patterns change frequently, the longer it takes for the system to process traffic data the less useful the results may be. For example, the system may choose a route where an accident has just occurred or avoid a clear route which was until recently jammed. Worse, malicious replicas may manipulate traffic data for their own purposes such as steering traffic to (or away from) particular locations or routes. If the application were built using a system that only offers eventual consistency, routing decisions could be based on inconsistent—and perhaps malicious—traffic data. However, building on traditional fault-tolerant consensus protocols such as Paxos or PBFT could result in a significantly less responsive system, undermining the application's usefulness.

One way of analyzing the conflict between responsiveness and consistency, as shown in Figure 1, is through the lens of fault detection. Consider an honest node which receives a travel time estimate from three replicas for Route A, as well as a travel time estimate from three other replicas for Route B. The node must choose which route to recommend before time t, otherwise the responder may not receive the recommended route in time to take it.

Now suppose that just before time t, only two (of three) estimates for Route A have arrived, but the estimates do not match. Furthermore, suppose that at most one of the three replicas could be faulty, so one of the estimates we received came from an honest node. Unfortunately, we cannot determine which one! The missing estimate could be from a faulty or honest node: the node may have crashed or the message may not have been delivered because of a network

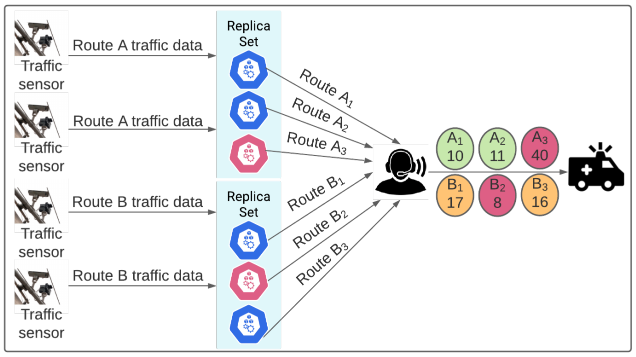

Figure 1. Route recommendation based on the proximal consensus outputs from two replica sets. Byzantine nodes in each replica set could attempt to influence the route choice using extremal values. By excluding outliers from the quorum, the dispatcher node makes more reliable recommendations.

outage. Furthermore, the two estimates we received could *both* be from honest nodes even though they differ: they may have been forced to use incomplete traffic data in their calculations if some sensor readings did not arrive in time.

An eventually consistent system might use the estimates anyway, perhaps "merging" the conflicting estimates by taking their average. Once the missing estimate arrives and the inconsistencies between the other estimates are resolved, the node will eventually produce a high-quality recommendation. Until then, however, it's unclear how reliable the node's recommendation is, or whether it should be used to make a safety-critical decision.

A traditional fault-tolerant system might not proceed until a sufficient number of messages are delivered to reach a majority consensus on the estimate. If a consensus is not reached within the allotted time, it could render the recommendation moot. Furthermore, traditional fault-tolerant systems cannot take advantage of workflows where approximate results are adequate for progress. Since a consensus cannot be reached unless a majority of replicas produce matching results, each honest replica must wait until all traffic data has been received before calculating their estimate to ensure it matches the estimates of other honest replicas.

This paper introduces *proximal consensus*, a new approach to building responsive, fault-tolerant consensus protocols for data-driven streaming applications with little-to-no coordination. Proximal consensus relies on *conditional probability distributions* that help diminish the impact extremal values (statistically unlikely outputs) have on the quorum's output. Furthermore, these distributions help indicate to clients the level of confidence they should place in the output. A high-confidence (high conditional probability) output indicates the output is likely to have come from honest (or honest-acting) replicas. Low confidence indicates that unexpected behavior occurred, either due to network delays or malicious replicas. To our knowledge, proximal consensus is the first consensus protocol to offer responsive, approximate outputs on streaming data without coordination.

Delivering outputs reliably, even when they have low confidence, allows applications to make domain-specific decisions when anomalous events occur. In our route-planning application, for example, the dispatcher node could choose to recommend routes based on older route estimates rather than update its recommendation based on new, low-confidence estimates.

In the remainder of this paper, we describe the core ideas behind proximal consensus (§2) and present Centauri, a system designed to use proximal consensus for responsive, fault-tolerant streaming applications (§3). We discuss proximal consensus in the context of related work (§4), and finally we discuss limitations and open questions (§5).

2 Proximal Consensus

What sets *proximal consensus* apart from other consensus protocols is that it does not rely on exact state machine replication. Instead, each host uses conditional probability distributions to determine the most likely result given the messages received before the deadline. The distributions are inferred from continuous, online empirical measurements by a trusted monitor and distributed asynchronously. This approach is attractive for our setting since it exhibits a kind of graceful degradation as faults accumulate or network conditions deteriorate. For example, whereas a BFT protocol during a network partition might be unable to reach consensus (possibly leading to a view change to select a new leader), a proximal consensus protocol could continue to make progress with the data available with reduced confidence in the results.

Honest nodes may be forced to produce outputs based on a subset of the data they expected to receive. Since significant network partitions are statistically unlikely in most systems, the smaller such a subset is, the less likely it will occur under benign conditions. Thus we can recognize extremal outputs based on their likelihood given probability distributions on the network latency and input values. There is no magic here though; just as BFT nodes often cannot distinguish between a faulty host and a network partition when an expected message is missing, proximal consensus nodes cannot always attribute the cause of an extremal output when one occurs. Instead, the key idea is that enabling the recognition of extremal outputs allows us to exclude them if a quorum of non-extremal outputs is available.

To explain proximal consensus in more detail, consider a simple coin flip application, illustrated in Figure 2, that counts the number of tails that have occurred. The result of each coin flip is sent to a replica set, which increments a counter for each tail. By using three replicas, the system

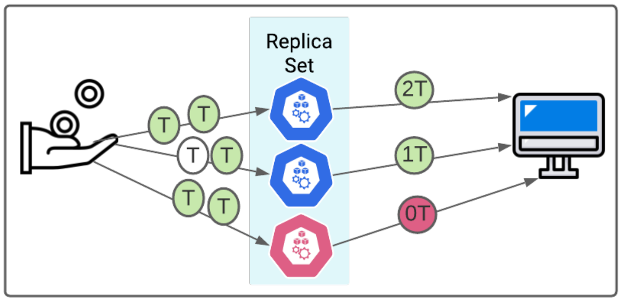

Figure 2. A set of $h = 2$ and $f = 1$ replicas that report the number of received tails (coin flip) at time t. The middle replica receives the second tails after its deadline to output.

designer hopes to be able to tolerate a crash or compromise of at most one of the nodes.

The top and bottom replicas receive two messages before the deadline, but the second message to the middle replica is delayed. Because each replica must produce a result before the deadline, the top replica sends 2T, but the middle replica sends 1T since it has only seen one tail result. The bottom node is Byzantine and sends 0T even though it has seen both messages. In order to determine the most likely output given these messages, the client must use a priori knowledge about how honest replicas behave and the probability distributions of system events.

2.1 System model

We assume that data streams flow from sources to clients in an acyclic dataflow graph. Edges in the dataflow graph represent network connections between source, replicas, or clients. Replicas, the interior nodes in the graph, perform computations on the streams of data they receive and produce an output stream for downstream consumers. Replica sets perform identical computations on the same streams, and send outputs to the same consumers. Each replica set is configured with a parameter f, which specifies the (assumed) maximum number of Byzantine faults tolerated by that set. For a replica set of size n configured to tolerate f faults, we assume at least $h = n - f$ replicas in the set are honest.

Clients and replicas have functions that encode the distributions of (the next element of) each input stream and the expected network latency of each connection. For example, the replicas in Figure 2 might have a uniform distribution $P(H) = \frac{1}{2}; P(T) = \frac{1}{2}$ for each coin flip, and an exponential distribution with arrival rate $\lambda = 20ms$ for the network delay.

2.2 Consensus as probability maximization

The goal of the client in Figure 2 is to determine the most likely number of tails based on its observations. As with traditional consensus protocols, proximal consensus attempts to form a quorum from a subset of replicas. Rather than finding a quorum of *matching* messages, however, the goal is to find the subset most likely to have been produced by honest replicas. In this way, elements from multiple *replica streams* can be used to construct a *logical stream* that is tolerant of Byzantine faults and network delays.

As a first step, consider the unconditional probability at the client of receiving 2T, assuming that the top replica is honest. An honest replica will only report 2T if it observes two T messages from the source. Therefore, we need the joint probability of two tail flips and two delivered messages. Let R_1 and R_2 be random variables for the time the first and second messages arrive at the replica. The probability of two tail flips is $P(\mathtt{TT}) = \frac{1}{2} \cdot \frac{1}{2}$. The probability of two delivered messages is the joint probability $P(R_1 \le t) \cdot P(R_2 \le t)$ where t is the deadline, giving us the joint probability $\frac{1}{4} \cdot P(R_1 \le t) \cdot P(R_2 \le t)$.

Now consider receiving 1T from an honest node. Here there are several scenarios. We could have TH or HT, or we could have TT and one of the messages fails to arrive in time:

$$\begin{aligned} P(1\mathtt{T}) = P(\mathtt{TH}) \cdot P(R_1 \le t) + P(\mathtt{HT}) \cdot P(R_2 \le t) \\ + P(\mathtt{TT}) \cdot (P(R_1 > t) + P(R_2 > t)) \end{aligned}$$

Finally, for receiving 0T from an honest node we have

$$\begin{aligned} P(0\mathtt{T}) = P(\mathtt{HH}) + P(\mathtt{TH}) \cdot P(R_1 > t) + P(\mathtt{HT}) \cdot P(R_2 > t) \\ + P(\mathtt{TT}) \cdot P(R_1 > t) \cdot P(R_2 > t) \end{aligned}$$

Formally, let X be a random variable representing the next element in the logical stream, and N_1, N_2, N_3 be random variables representing messages received from the top, middle, and bottom nodes, respectively. Let q be a subset of size $n - f = 2$ of the observed events $Q = \{N_1 = 2\mathtt{T}, N_2 = 1\mathtt{T}, N_3 = 0\mathtt{T}\}$. The client's task is to find a pair (x, q) such that the conditional probability $P(X = x \mid q)$ is maximized:

Definition 2.1 (Proximal consensus of X given Q)**.**

$$\mathbf{PC}(X; Q) \triangleq \underset{x\,;\,q \in [Q]^{n-f}}{\operatorname{argmax}} \; P(X = x \mid q)$$

In other words, find the most likely output over all valid quorums of size $n - f = 2$.

A key idea of our approach is that since all replicas are expected to send outputs by a pre-set deadline, clients (or downstream replicas) can "observe" when a message is not received by the expected time. Suppose that the output 1T from the middle replica in Figure 2 failed to arrive at the client by the expected time. Here, the observed events would instead be $N_1 = 2\mathtt{T}, N_2 = R_2 > t, N_3 = 0\mathtt{T}$. The client did not observe a message from the middle replica (N_2), but it did observe that the arrival time of any such message will be greater than the deadline t.

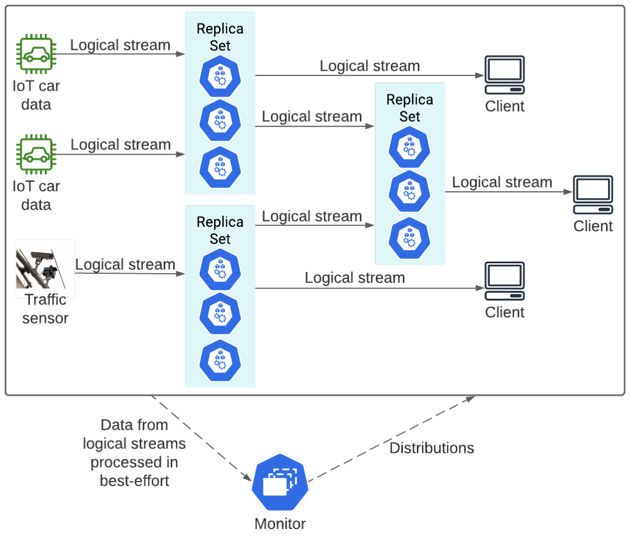

Figure 3. Centauri architecture. Solid arrows represent the receiving node asynchronously processes received data on streams. Dotted arrow signify data is processed and transmitted in best effort.

3 Centauri Architecture

3.1 System Model

Figure 3 shows the five components of Centauri, a framework for responsive data streams that uses proximal consensus:

- **Data sources** (e.g., stationary and mobile IoT devices), which produce and stream data to an endpoint.
- **Replicas**, untrusted servers that process and store data from subscribed streams. Replicas run computations on data and stream results to specified endpoints.
- **Replica Set**, a set of $h + f$ replicas subscribed to the same set of endpoints, where h and f is the assumed number of honest and Byzantine replicas. Each replica in a replica set runs the same time-window computation at a predetermined interval and asynchronously processes received data.
- **Monitor**, collects insights on all data streams, computes statistical distributions on the data, and provides distributions to replicas and clients in best effort. The distributions enable clients to produce confidence levels on results produced from proximal consensus.
- **Clients**, end-users that issue requests on data and subscribe to sets of endpoints. They have limited computing resources and interact with the system using an application on a limited resource device such as a cellphone or laptop.

The monitor provided distributions enables a node to run proximal consensus. Specifically, the distributions enable the client to take a set of inputs (replica computations on data) and the amount of assumed Byzantine replicas (f), and

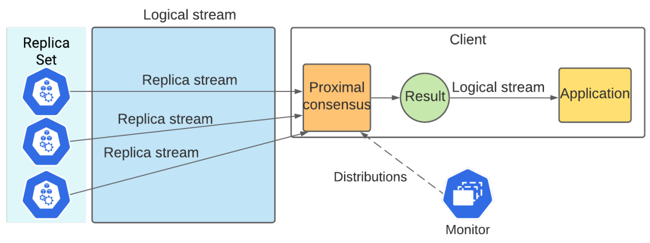

Figure 4. The flow of untrusted data streams from a replica set to a client, who uses the (potentially) partial results to infer a single value.

output a statistically likely result, along with an associated confidence. The result and confidence level is then streamed into the application as a trusted input. The proximal consensus process is shown in Figure 4.

3.2 Message Types

Proximal consensus contains four types of messages: request, stream, sample, and distribution.

Request messages are sent from clients to replicas and replicas to replicas. The messages are used to set up a set of *replica streams* of some continuous computation on data over a defined period. Request messages appear as such: $\langle type = \texttt{request}, id, req, v, ttl, r\rangle_\sigma$, where:

- id, the identifier of the requesting node.
- req, the request.
- v, the validity of data items for each computation satisfying the predicate: src_time $\geq now() - v$.
- ttl, how long to run the continuous query.
- r, the rate at which to produce updates (e.g. every four seconds).
- σ, the sender's signature.

Stream messages are received by nodes subscribed to endpoints and are sent from data sources and replicas. This is commonly from data sources to replicas, and replicas to clients. Stream messages are in the form: $\langle type = \texttt{stream}, id, s_{id}, ts, n, D\rangle_\sigma$, where:

- id, the identifier of the sending node.
- s_{id}, the stream id.
- n, the monotonically increasing nonce of the message on s_{id}
- ts, the source timestamp of the message.
- D, the produced data at time ts.
- σ, the sender's signature.

Sample messages provide the monitor with empirical data to infer probability distributions. There are two sub types of sample messages. The first sub type, o, is from data sources and replicas and contain output values (raw data or computations on data). The second sub type n is from replicas and clients and contain network latencies measured

by the difference between the source timestamps and the time the message was received. Sample messages are in the form:
$\langle type = \texttt{sample}_{\texttt{o|n}}, d, D, ts\rangle_{\sigma}$, where:

- o|n, specifies the sub type.
- d, the identifier of the distribution associated with the data.
- D, the data. For type o samples, this is the output value. For type n samples, this is a network latency.
- ts, the timestamp of the sample message.
- σ, the sender's signature.

Distribution messages are sent to replicas and clients containing the distributions on the data of interest. Distribution messages are in the form:
$\langle type = \text{distribution}, d, \Delta, ts\rangle_{\sigma}$, where:

- d, the identifier of the distribution.
- Δ, a vector containing the distribution type and its parameters.
- ts, the timestamp of the message.
- σ, the sender's signature.

4 Related Work

To the best of our knowledge, proximal consensus is the first consensus protocol to form quorums with partial replica state and no replica synchronization. By learning the context of the data, proximal consensus can produce distributions and analyze data in real-time. Traditional Byzantine agreement protocols, such as PBFT [9] and Paxos [19] are general consensus solutions, where data is processed in total order and accepted only if a quorum of replicas produce matching results. Zyzzyva [16], HotStuff [29], BFT-SMART [7], to list a few, offer optimizations to PBFT and Paxos by optimizing the process of totally ordering messages (i.e. state machine replication). Although these protocols do not need to reason about the data (why they are general), generality comes with unnecessary overhead.

Gupta et al. [15] and Kardam [11] propose solutions to be robust against Byzantine agents in distributed stochastic gradient descent (D-SGD) As far as we know, the only body of work that is similar to us is in Byzantine machine learning.

Much of the previous work in streaming systems offer trade offs of availability for consistency or other eventually consistent guarantees, which makes it impossible to estimate the accuracy of the produced result. For example, Aurora [2] provided a stream-oriented solution for monitoring continuous window queries with asynchronous message arrival, but computations on data is trusted and offers no fault-tolerance. Borealis [1] extends Aurora and offers a method for fault-tolerance through a dynamic revision of query results for delayed data items by offering eventually consistent guarantees; however, it is impossible to predict the skew of the initial computation. Balazinska et al. [5] and CEDR [6] provide eventually consistent guarantees; however, it is impossible to determine how inaccurate the results are, which are also issues with Apache Storm at Twitter [26] and Twitter Heron [17] (extending Storm). Rhino [13] fails to support out-of-order (asynchronous) data processing as records are assumed to be timestamped with a monotonically increasing logical timestamp, which is against our desire for responsiveness. Apache Spark Streaming [30] relies on checkpoints to ensure consistency and fault-tolerance if a working node fails, but the solution requires the use of extra resources. Macrobase [4] is a data analytics engine that uses machine learning to explain fast data volumes to users, but does not offer fault-tolerance or mitigation against malicious data inputs.

5 Discussion

Proximal consensus is a protocol designed to maintain responsiveness even when a quorum fails to form on matching responses. Current consensus protocols may choose to synchronize replica state or attempt a view change to replace a faulty primary, but synchronization causes the system to block until replicas finish synchronizing state. Alternatively, proximal consensus aims at responsively outputting replica computations even if replicas produce non-matching outputs. Using the monitor-provided distributions, the client can identify the $n - f$ most statistically likely replica outputs and infer the most statistically correct answer given the observed outputs, along with computing the confidence level of the final result. Reducing the impact f of the most outlying responses have on the final result enables proximal consensus to limit the impact Byzantine responses may have on the final result.

This paper has given an overview of proximal consensus and the challenges of maintaining responsiveness in quorum-based systems when replicas produce inconsistent results. We find that the accuracy of the monitor-produced distributions is essential to identify honest replicas and produce high-confidence results correctly. While there are some approaches to estimate a distribution on a set of inputs, such as Maximum Likelihood Estimation [21] or The Bayesian Way [22], there are exciting opportunities to robustifying these estimation techniques. Robustifying estimations is essential for the monitor to produce distributions where some inputs may be from malicious replicas [24, 27, 28]. Techniques in Federated Learning, and more recently, Cluster Federated Learning, are interesting mechanisms to explore in building Byzantine-robust learning models [14, 25]. Moreover, robust Bayesian networks [10] can also play a suitable role in using conditional probability to determine the likelihood of an event occurring given a set of (partial) observed events with the presence of Byzantine adversaries. We see the aforementioned techniques as methods for the monitor to build distributions capable of accurately detecting anomalies (low probability events) on partial state while also minimizing the adverse impact of Byzantine adversaries.

Responsive-minded systems that run continuous queries on stable data streams with a known statistical distribution are strong candidates to leverage proximal consensus. Systems with non-stable data flows containing no known statistical distribution are more challenging to support because of the difficulty of using conditional probability to identify the possibly malicious replica.

References

[1] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, et al. 2005. The design of the borealis stream processing engine.. In *Cidr*, Vol. 5. 277–289.

[2] Daniel J Abadi, Don Carney, Ugur Cetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. 2003. Aurora: a new model and architecture for data stream management. *the VLDB Journal* 12, 2 (2003), 120–139.

[3] J Chris Anderson, Jan Lehnardt, and Noah Slater. 2010. *CouchDB: the definitive guide: time to relax.* " O'Reilly Media, Inc.".

[4] Peter Bailis, Edward Gan, Samuel Madden, Deepak Narayanan, Kexin Rong, and Sahaana Suri. 2017. Macrobase: Prioritizing attention in fast data. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 541–556.

[5] Magdalena Balazinska, Hari Balakrishnan, Samuel Madden, and Michael Stonebraker. 2005. Fault-tolerance in the Borealis distributed stream processing system. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. 13–24.

[6] Roger S Barga, Jonathan Goldstein, Mohamed Ali, and Mingsheng Hong. 2006. Consistent streaming through time: A vision for event stream processing. *arXiv preprint cs/0612115* (2006).

[7] Alysson Bessani, Joao Sousa, and Eduardo EP Alchieri. 2014. State machine replication for the masses with BFT-SMART. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 355–362.

[8] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36, 4 (2015).

[9] Miguel Castro, Barbara Liskov, et al. 1999. Practical byzantine fault tolerance. In *OSDI*, Vol. 99. 173–186.

[10] Guangjie Chen and Zhiqiang Ge. 2020. Robust Bayesian networks for low-quality data modeling and process monitoring applications. *Control Engineering Practice* 97 (2020), 104344.

[11] Georgios Damaskinos, Rachid Guerraoui, Rhicheek Patra, Mahsa Taziki, et al. 2018. Asynchronous Byzantine machine learning (the case of SGD). In *International Conference on Machine Learning*. PMLR, 1145–1154.

[12] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS operating systems review* 41, 6 (2007), 205–220.

[13] Bonaventura Del Monte, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2020. Rhino: Efficient management of very large distributed state for stream processing engines. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2471–2486.

[14] Minghong Fang, Xiaoyu Cao, Jinyuan Jia, and Neil Gong. 2020. Local model poisoning attacks to byzantine-robust federated learning. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*. 1605–1622.

[15] Nirupam Gupta, Shuo Liu, and Nitin Vaidya. 2021. Byzantine Fault-Tolerant Distributed Machine Learning with Norm-Based Comparative Gradient Elimination. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE, 175–181.

[16] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. 2007. Zyzzyva: speculative byzantine fault tolerance. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*. 45–58.

[17] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M Patel, Karthik Ramasamy, and Siddarth Taneja. 2015. Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 239–250.

[18] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review* 44, 2 (2010), 35–40.

[19] Leslie Lamport et al. 2001. Paxos made simple. *ACM Sigact News* 32, 4 (2001), 18–25.

[20] MongoDB. [n.d.]. MongoDB: Agile and Scalable. http://www.mongodb.org/.

[21] In Jae Myung. 2003. Tutorial on maximum likelihood estimation. *Journal of mathematical Psychology* 47, 1 (2003), 90–100.

[22] Svein Olav Nyberg. 2018. *The Bayesian way: introductory statistics for economists and engineers*. John Wiley & Sons.

[23] Daniel Peng and Frank Dabek. 2010. Large-scale incremental processing using distributed transactions and notifications. (2010).

[24] Xiaoqiang Ren, Yilin Mo, Jie Chen, and Karl Henrik Johansson. 2020. Secure state estimation with byzantine sensors: A probabilistic approach. *IEEE Trans. Automat. Control* 65, 9 (2020), 3742–3757.

[25] Felix Sattler, Klaus-Robert Müller, Thomas Wiegand, and Wojciech Samek. 2020. On the byzantine robustness of clustered federated learning. In *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 8861–8865.

[26] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. 2014. Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 147–156.

[27] Zhixiong Yang, Arpita Gang, and Waheed U Bajwa. 2019. Adversary-resilient inference and machine learning: From distributed to decentralized. *stat* 1050 (2019), 23.

[28] Zhixiong Yang, Arpita Gang, and Waheed U Bajwa. 2020. Adversary-resilient distributed and decentralized statistical inference and machine learning: An overview of recent advances under the Byzantine threat model. *IEEE Signal Processing Magazine* 37, 3 (2020), 146–159.

[29] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. 2019. HotStuff: BFT consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*. 347–356.

[30] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion Stoica. 2012. Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In *4th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 12)*.