High-Availability at Massive Scale: Building Google's Data Infrastructure for Ads

Ashish Gupta and Jeff Shute

Google Inc. {agupta,jshute}@google.com

Summary. Google's Ads Data Infrastructure systems run the multi-billion dollar ads business at Google. High availability and strong consistency are critical for these systems. While most distributed systems handle machine-level failures well, handling datacenter-level failures is less common. In our experience, handling datacenter-level failures is critical for running true high availability systems. Most of our systems (e.g. Photon, F1, Mesa) now support multi-homing as a fundamental design property. Multi-homed systems run live in multiple datacenters all the time, adaptively moving load between datacenters, with the ability to handle outages of any scale completely transparently.
This paper focuses primarily on stream processing systems, and describes our general approaches for building high availability multi-homed systems, discusses common challenges and solutions, and shares what we have learned in building and running these large-scale systems for over ten years.

Key words: stream processing, distributed systems, multi-homing, databases

1 Introduction

Google's Ads platform serves billions of advertisements every day to users around the globe, and generates a large volume of data capturing various user interactions, e.g., seeing an Ad, clicking on an Ad, etc. The Ads Data Infrastructure team is responsible for processing and managing all this data in near real-time, and delivering critical reports and insights to Google's Ad users and clients. This includes generating rich reports for our advertisers on how their Ad campaigns are performing, managing budget in the live serving system, etc. Reliable and timely delivery of this data is critical for Google and its partners, and for supporting Google's multi-billion dollar Ads Business. Consistency is also a strong requirement since the advertising data determines revenue and billing, and because inconsistent data is very confusing and hard to deal with, both for end users and for developers. In this paper we explore what it takes to build such mission-critical systems while ensuring consistency and providing high availability, particularly in the context of real-time streaming systems.

Over several years and multiple generations, our strategies for high availability have evolved considerably. Our first generation systems focused primarily on handling machine-level failures automatically. Although datacenter failures are rare, they do happen, and can be very disruptive, and are especially difficult to recover from when data consistency is lost. Our next generation systems were designed to support datacenter failovers and recover state cleanly after failover. Furthermore, we implemented automated failure detection and failover, which reduced operational complexity and risk. Failover-based approaches, however, do not truly achieve high availability, and can have excessive cost due to the deployment of standby resources. Our current approach is to build natively multi-homed systems. Such systems run hot in multiple datacenters all the time, and adaptively move load between datacenters, with the ability to handle outages of any scale completely transparently. Additionally, planned datacenter outages and maintenance events are completely transparent, causing minimal disruption to the operational systems. In the past, such events required labor-intensive efforts to move operational systems from one datacenter to another.

We've recently published details of several of these large-scale systems (e.g. F1 [26], Mesa [21], and Photon [3]), all of which require large global state that needs to be replicated across multiple datacenters in real-time. F1 [26] is a distributed relational database system that combines high availability and the scalability of NoSQL systems like Bigtable [12], and the consistency and usability of traditional SQL databases. Photon [3] is a geographically distributed data processing pipeline for joining multiple continuously flowing streams of data in real-time with high scalability, low latency, strong consistency (exactly-once semantics), and high reliability. Finally, Mesa [21] is a petabyte-scale data warehousing system that allows real-time data ingestion and queryability, as well as high availability, reliability, fault tolerance, and scalability for large data and query volumes.

The rest of this paper is organized as follows. In Section 2, we state our assumptions and model, and what we mean by availability and consistency, particularly in the context of streaming systems. Section 3 provides the background about some of the common failure scenarios based on our experience with operating these systems. In Section 4, we describe the different availability tiers, followed by the challenges in building multi-homed systems in Section 5. Section 6 describes some of the multi-homed systems built for Google's Ads systems. Section 7 discusses related work and Section 8 summarizes our conclusions.

2 Availability and Consistency for Streaming Systems

2.1 Assumptions and Context

In any of our typical streaming system, the events being processed are based on user interactions, and logged by systems serving user traffic in many datacenters around the world. A log collection service gathers these logs globally and copies them to two or more specific logs datacenters. Each logs datacenter gets a complete copy of the logs, with the guarantee that all events copied to any one datacenter will (eventually) be copied to all logs datacenters. The stream processing systems run in one or more of the logs datacenters and processes all events. Output from the stream processing system is usually stored into some globally replicated system so that the output can be consumed reliably from anywhere.

In this paper, we use datacenter to mean a cluster of machines in the same region. As a simplification, we assume all datacenters are geographically distributed.

In these streaming systems, it is important to note that processing is often not deterministic and not idempotent. Many streaming pipelines have complex business logic with stateful processing, with order and timing dependencies. For example, identifying sessions and detecting spam events both require clustering events and are highly dependent on which events have been processed so far.

2.2 Availability for Streaming Systems

In most contexts, a system is available if it is alive and able to accept user requests. In a streaming system, availability has a different meaning. It is not critical that the system actually be responsive millisecond by millisecond. Instead, the system must stay alive over time and continue processing events. Typically, availability will be measured based on overall delay in the streaming system.

Availability targets are often expressed as a percentage availability over time. In a streaming system, a target could be that 99.9% of input events are fully processed in less than three minutes after arrival, and we consider the system available if this target is met for 99% of the time in a quarter. This system would not be meeting its availability goal if many events are delayed more than three minutes for long time intervals. Note that an availability target of 99% maps to 22 hours of downtime every quarter, which is too high for mission critical systems. Our systems often target 4 to 5 nines of availability.

2.3 Consistency for Streaming Systems

Consistency in streaming systems is analogous to consistency in databases. The ACID properties are the fundamental consistency expectations in a database. In a streaming system, the core consistency requirement is exactly once processing, i.e., each input event must be processed *exactly once*. Both systems have some additional consistency properties about visibility of outcomes. In a database, a user that commits a transaction and then does a query should see the result of the previous write. In a streaming system, a user observing state repeatedly should see the state moving forward, i.e., if a state at time t includes an event e then any future state at time $t' \geq t$ must also include event e. In both cases, where there are multiple stored outcomes, an observer should see consistent outcomes. For example, on any screen with multiple reports or multiple graphs, users expect the totals to match.

It is possible to view a streaming system as a database processing micro-transactions, one per event, with the same expectations of transactional consistency that would be expected in a database. The key difference is that a streaming system is optimized to run these micro-transactions at extreme scale, using techniques like *batching* to optimize for throughput rather than latency.

Streaming systems can be used to solve many problems, only some of which require consistency. Many streaming systems are used to scan a stream of events and compute some approximate summary [20, 25, 24, 16, 17], based on a transient time window, with the summary used primarily for display, or as a coarse input signal to another system. A common example in many streaming systems is an application that extracts user sentiment from recent tweets in the Twitter stream. Consistency is not critical in such systems. If events are lost or inconsistent at some point, it won't matter once the output moves past that point and shows current data.

In other cases, streaming systems are required to process each event exactly once, and generate some exact and permanent record of the set of events observed. The system may make automated decisions based on the observed stream, which may have permanent side-effects. For example, an advertising system may be tracking user clicks on an ad, making decisions about which events are spam, deciding how much to charge for each click, accumulating an invoice and then charging an advertiser's credit card. In these cases, all decisions must be permanent, and side-effects cannot be lost.

In this paper, we are considering streaming systems where consistency is paramount.

3 Failure Models

3.1 Typical Failure Scenarios

There are many classes of failures in a distributed system:

Machines In a large datacenter, individual machine failure occurs frequently, and can be caused by hardware or software issues. Multi-machine failures like power failures can also happen at the rack or datacenter level, but are much rarer.

Network connectivity Network connectivity issues also happen, within a datacenter and between datacenters. In rare circumstances, network partitions can happen that separate some datacenters completely from the rest of the wide area network. Partial failures are more common, with effects like reduced bandwidth, increased latency, unreliable connections, asymmetric connectivity, and unexplained flakiness. These failures can be triggered by hardware issues or by software issues, like throttling caused by network bandwidth over-subscription.

Underlying infrastructure Complex distributed systems are often built in layers, where the user-facing system has several underlying infrastructure components like distributed filesystems, databases, and communications services. Building on top of existing infrastructure allows reuse and simplifies systems, but it also means systems are exposed to failures in underlying infrastructure. In particular, with shared services, isolation between users can be very challenging, and if a shared system is over-subscribed, some users will see compromised performance.

3.2 Planned vs Unplanned Failures

Failures can be either *planned* or *unplanned*. Planned outages happen when machines, networks, datacenters, or software systems are taken offline for hardware changes, kernel upgrades, reconfiguration, software releases, etc. Before a planned outage, systems have the opportunity to gracefully shut down, drain any in-flight work, generate checkpoints, and resume in an alternate datacenter. Unplanned failures provide no opportunity for clean shutdown, and no general mechanism to discover the state of a system before it failed.

3.3 Partial vs Full Failures

Complete unplanned failure of a datacenter is a rare occurrence that may happen only once every several years. *Partial failures* affecting some significant fraction of resources in a datacenter are much more common. During a partial failure, a system may have fewer working machines, may have less bandwidth and higher latency, and components may have reduced reliability. In general, systems will run slower and have less capacity, and may see elevated failure rates and more timeouts. Streaming system may not be able to keep up with the input stream, and may fall behind.

While partial failures are more common than total failure, they are also harder to detect, diagnose and recover from. The problem often cannot be detected until after a system has already fallen behind, and then an engineer must try to diagnose which component is slow and why. In a complex system with many feedback loops, this can be very difficult because the component with the visible backlog may not be source of the problem.

Once a problem is diagnosed, it is often unclear how long a fix will take. An operations team may provide estimates of when a networking problem can be repaired or an infrastructure issue can be mitigated, but fixes may take longer, or may not work on the first try.

Uncertainty about fixes can put teams in a difficult position. For example, in a common scenario, the primary datacenter for a system is compromised, a fix is estimated to take one hour, and failover to a secondary (less desirable) datacenter is possible but will take 90 minutes, and requires a complex procedure which includes some risk. In this situation, the team must make a difficult decision about whether to wait out the outage or trigger a failover, with incomplete information about the outcome of either option.

4 Availability Tiers

To ensure high-availability in the case of a datacenter level outage, there are several possible approaches. Each approach has different trade-offs in availability SLAs, consistency guarantees, resource usage, etc. We categorize these approaches into different availability tiers.

4.1 Singly-homed systems

Singly homed systems are designed to primarily run in a single datacenter. In case of an intermittent datacenter outage, processing gets delayed. If the datacenter is unavailable for an extended period, the system must be restarted from some arbitrary point in another datacenter. This results in extended unavailability, as well as potential loss of data and consistency.

Singly-homed systems are relatively simple to build and are a good solution when inconsistencies or data losses are acceptable. One simple technique used to achieve high availability is to run a singly-homed system in parallel in multiple datacenters, and either allow user queries to go to any datacenter, or designate one datacenter as the primary that receives all user traffic, with alternate datacenters available as hot standbys that can be made into the new primary quickly. This approach does not require inter-datacenter coordination, but this simplification comes at the expense of consistency guarantees between datacenters.

4.2 Failover-based Systems

In failover-based systems, processing still happens in a single primary datacenter, but the critical system state capturing what has been processed is replicated as a *checkpoint* in an alternate datacenter. In case of a primary datacenter outage, processing can be moved to the alternate datacenter and restarted from the latest checkpoint.

Checkpoints can be used in two ways. In the simple case, a checkpoint is taken asynchronously and periodically, expressing what work has already been completed. This checkpoint provides an approximate restart point that will ensure all events still get processed at least once, without excessive reprocessing of already-completed events. These asynchronous checkpoints are sufficient to preserve exactly once processing during planned outages; in-progress work can be drained before taking a checkpoint, and the checkpoint can be used to restart the pipeline in an alternate datacenter from exactly where it left off.

More advanced systems can be built where checkpoints are taken synchronously and describe exactly what has been processed. At some point during processing, the pipeline generates a checkpoint and blocks until the checkpoint is replicated before proceeding. Synchronous checkpoints can exactly capture the processing state, even for unplanned failovers, but the systems are typically much more complex since checkpoints must be tied directly into the processing and recovery pipelines.

With large complex systems, failover procedures (with synchronous or asynchronous checkpoints) tend to be very complex and risky. Complex systems have many components and many dependencies, all of which need to restarted and potentially reconfigured. Manual failovers means handling each component individually, which is time consuming and error-prone.

Our teams have had several bad experiences dealing with failover-based systems in the past. Since unplanned outages are rare, failover procedures were often added as an afterthought, not automated and not well tested. On multiple occasions, teams spent days recovering from an outage, bringing systems back online component by component, recovering state with ad hoc tools like custom MapReduces, and gradually tuning the system as it tried to catch up processing the backlog starting from the initial outage. These situations not only cause extended unavailability, but are also extremely stressful for the teams running complex mission-critical systems.

It is usually clear that failover should be automated to make it faster and safer, but automation has its own challenges. In complex systems, failover processes are also complex and have many steps. Furthermore, as systems evolve, failover scripts must be updated with every change. Keeping scripts up to date can be challenging, and real-life failovers are often not well tested because the full process cannot be exercised without causing an outage.

With effort, failovers can be successfully automated and made to run relatively quickly, but there are still inherent problems. First, during an outage, a team has to make difficult decisions with incomplete information about the nature of an outage and the time until it will be resolved (See Section 3.3). Second, failover procedures are still complicated, and must still be maintained with every change and regularly tested on live systems. This imposes a significant burden on a team's development bandwidth, and makes failover-based systems inherently high-maintenance.

4.3 Multi-homed Systems

Many of the problems with failover-based systems stem from the fact that failover is usually built as an add-on feature on top of an inherently singly-homed design, adding complex new behavior that is not part of the primary execution path.

In contrast, multi-homed systems are designed to run in multiple datacenters as a core design property, so there is no on-the-side failover concept. A multi-homed system runs live in multiple datacenters all the time. Each datacenter processes work all the time, and work is dynamically shared between datacenters to balance load. When one datacenter is slow, some fraction of work automatically moves to faster datacenters. When a datacenter is completely unavailable, all its work is automatically distributed to other datacenters. There is no failover process other than the continuous dynamic load balancing.

Multi-homed systems coordinate work across datacenters using shared global state that must be updated synchronously. All critical system state is replicated so that any work can be restarted in an alternate datacenter at any point, while still guaranteeing exactly once semantics. Multi-homed systems are uniquely able to provide high availability and full consistency in the presence of datacenter-level failures. Building multi-homed systems, however, poses novel challenges, which we explore in the next section.

5 Challenges in Building Multi-homed Systems

5.1 Synchronous Global State

In order to process each work unit exactly once, and support unplanned failover to an alternate datacenter, the state of work units must be stored globally. If work has been completed in one datacenter, the system must not lose that state after a failover because that could lead to double-processing.

Maintaining global state consistently and reliably introduces significant latency. In general, processing datacenters are geographically distributed to avoid correlated outages such as power failures, network failures, or natural disasters. Typically, round trip network latency between geo-distributed datacenters is at least tens of milliseconds. Thus updating global state synchronously takes at least that long. We use Paxos-based commit to update metadata synchronously [22]. In most cases, this means storing metadata in Spanner[15], which acts as a globally replicated database with synchronous commit.

The latency involved in synchronous commits makes it necessary to design systems for maximal throughput despite high latency of individual operations. Systems must support high parallelism, and typically use some form of batching to reduce round trips to global state. Serial operations using global state must be avoided as much as possible. In many cases, pipelines can be arranged so several operations can be clustered together and applied as a group locally, with global state updates only at the boundaries between operation clusters.

Global metadata commits also require wide area network bandwidth, which is more expensive than local bandwidth. The size of global state should be minimized to limit this bandwidth usage as much as possible. Global state should usually be used for metadata only, and not the data itself. Additionally, work should be batched, and batches of work should be represented with small state when possible. For example, when input comes from files, describing a batch with a byte range of an input file is much more compact than storing a list of event IDs included in the batch (as long as the same input file will be reproducible in alternate datacenters).

5.2 What to Checkpoint

A complex processing pipeline may have multiple inputs, multiple outputs, and many internal nodes. As data flows through the pipeline, each node has some state (i.e. data), and some metadata (e.g. queues of pending and completed work). Some nodes may produce deterministic output based on their input, but some nodes may be non-deterministic. For example, a node may do lookups into a mutable database, or may perform some time-sensitive or order-sensitive computation (e.g. clustering events into sessions, identifying the first unique occurrence of an event, etc.).

As discussed earlier, checkpointing to global state has high cost in latency and bandwidth, so it is desirable to minimize the number and size of checkpoints. Checkpointing can be minimized by clustering processing nodes together and only checkpointing at the endpoints of the cluster. Within the cluster, all state should be kept local within a datacenter.

Clustering checkpoints requires understanding what state exists at each point, and determining what state can be regenerated after a failure. Any deterministic state can be trivially recomputed, but non-deterministic state can also be regenerated (possibly with different results) as long as no output was produced and no observable side-effects have happened.

In the best case, a system may be able to run with no checkpoints at all, up to the point of final output, where the global state is used to determine whether some particular output has already happened. More typically, systems do initial checkpointing at the input stage of the pipeline to record what events were batched together, so later state can be recorded per-batch, and so failover and re-processing can happen efficiently at the batch level. In complex pipelines with multiple outputs and side-effects, it is often necessary to checkpoint state at intermediate nodes where the output of some node will be used as input to multiple downstream nodes leading to multiple different final outputs.

5.3 Repeatable Input

Storing small checkpoints selectively at particular nodes is a significant optimization, as discussed above. One key property that makes that optimization possible is repeatability of input. If the same input data will be available in multiple datacenters, then a checkpoint can describe input using a pointer to the input data (e.g. an event ID or an offset in a file) rather than storing a copy of the input itself. If the input data is not itself multi-homed, then a checkpoint in global state will not be useful unless it contains a full copy of the input data, which could be very expensive. In many of our systems, the input comes from event logs, which are collected by a logging system that ensures that all log events are (eventually) copied to two separate datacenters.

Many processing pipelines also have secondary inputs, where data from other sources is joined with the event data from the primary source. For example, database lookups are common. Logged events typically include object IDs but not the full state of the object. The business logic computed in the pipeline often references other attributes of the referenced objects, which requires a database lookup. If these lookups are not inherently deterministic, additional checkpointing is more complicated.

Some database lookups are inherently repeatable. For example, looking up the country name where an event occurred should be deterministic because the set of countries and their attributes is immutable in the database. Any lookups of objects that are insert-only and immutable and guaranteed to exist can also be treated as deterministic.

Any database lookup for a mutable object will not be repeatable. For example, a currency rate will change frequently. Most data objects store their current state only, which can be updated (or deleted) at any time by a user or by other business logic, changing the results of future lookups. To reduce checkpoint cost, sometimes it is possible to make these lookups repeatable. For example, a database may support snapshot reads as of a timestamp, or the schema may be designed to record history of changes. In either case, if each event has a timestamp, database lookups can retrieve the state as of that timestamp.

Where a lookup cannot be made repeatable, the result of the lookup must be stored as part of any checkpoint stored post-lookup so that if the pipeline resumes from the checkpoint, it will have the same starting point. To avoid storing this state, it may be beneficial to push non-deterministic lookups near the end of the pipeline, when possible.

5.4 Exactly Once Output

As mentioned earlier, a key semantic property is that each input event should be processed exactly once. This can be challenging to guarantee in a multi-homed system because pipelines may fail at any point while producing output, and multiple datacenters may try to process the same events concurrently. Distributed systems typically employ backup workers [18] attempting duplicate work to reduce tail latency, which means that even without any failures, multiple machines may try to process the same data concurrently. All outcomes from that processing must be observable exactly once in the visible output.

Ideally, updating global state should be atomic with the actual output operation of the pipeline. This is possible when the stream metadata and final output are stored in a shared storage system that supports transactions across entities, making clean handoff from the streaming pipeline to the output system easy, and providing exactly-once trivially. Shared transactions like this are often not possible, however, since the pipeline and output systems do not usually share the same infrastructure or the same databases. There are two possible solutions to ensure exactly-once processing in such cases.

Idempotent Output When possible, idempotence can provide exactly-once guarantees. The processing system writes output first, and then commits global state to record completion. If another datacenter processes the same work, it will repeat the same writes, which will have no effect. It is critical that all input and all processing be deterministic so that duplicate processing actually produces duplicate output that can be elided by idempotence.

Some outcomes are naturally idempotent. For example, some systems generate notification events where duplicate notifications have no side-effect in the receiver. When output is written into an insert-only storage system, where each record has a key and there are no deletions or mutations, writes can be made idempotent by ignoring or overwriting on key collisions. Note that if another system consumes records from this output and then updates or deletes the record, the writes will be no longer idempotent.

Some outcomes are not inherently idempotent but can be made idempotent. For example, applying an arbitrary transaction to a database will not be idempotent, but if each transaction can be assigned a deterministic ID, and the processed IDs can be inserted transactionally in a separate table in the same output database, then checking presence of the target ID will make the transaction idempotent.

Two-Phase Commit When a system's output cannot be made idempotent, it is often necessary to use some form of two-phase commit to atomically move output from the streaming system into the output system. The general protocol commonly looks like

1. Record decision to commit a batch of work in the source system.
2. Write final output into the target system and commit. (It must be possible to inspect whether this has happened.)
3. Record completion of the commit in the source system.

The streaming system drives commit with a state machine, and after a restart or any operation with indeterminate output, the current state can be retrieved by inspection and the state machine can resume cleanly and still provide exactly-once handoff.

5.5 Resource Cost

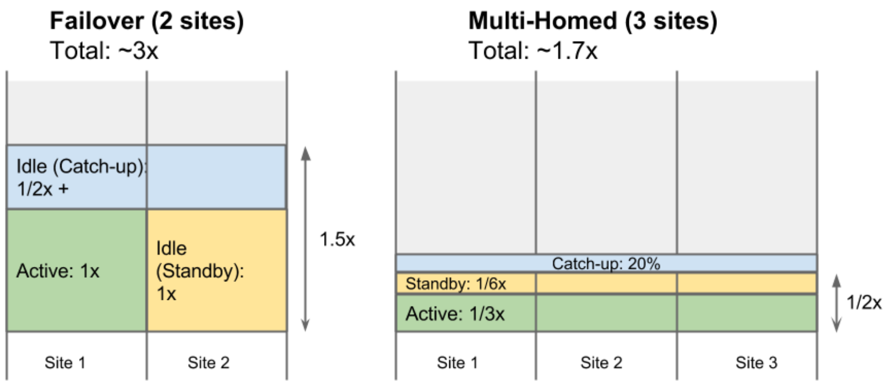

Fig. 1. Resource Cost for failover-based and multi-homed systems

Running a multi-homed pipeline means having processing capacity in multiple datacenters. Perhaps surprisingly, the resource requirements can actually be significantly less than would be required to run singly homed pipelines with failovers.

Consider a failover-based system running with two datacenters - one live and one on standby. The live datacenter needs capacity to process 100% of the traffic, and the idle datacenter must have equal potential capacity so it could take over after failover. It is not sufficient to have exactly 100% of required steady state capacity, however. There must be sufficient capacity to handle load spikes, and there must be sufficient capacity to catch up after delays or failures. Because partial failures are common and can be hard to diagnose, a datacenter may run with reduced performance for some time before the datacenter recovers or failover happens, which will lead to large delays. A system then needs to be able to process data faster than real-time in order to catch back up, which requires proportional extra capacity. For example, if a system has only 10% extra capacity, a one-hour delay would take ten hours to catch up, and recovering from multi-hour delays would take multiple days.

Deploying our systems with at least 50% extra capacity for catch-up and recovery is typical, which means the total resource footprint is 300% of steady state, because extra capacity needs to be available in whichever datacenter is live at the time.

In a multi-homed system, all datacenters are live and processing all the time. Deploying three datacenters is typical. In steady state, each of the three datacenters process 33% of the traffic. After a failure where one datacenter is lost, the two remaining datacenters each process 50% of the traffic. As before, additional capacity is needed to handle spikes and to catch up after any delay. The catch-up capacity requirement is lower, for several reasons:

- During catch-up, all remaining datacenters can contribute work, so less work is required per datacenter, and no standby catch-up capacity is wasted.
- The system continuously load balances, including during partial failures, so when a datacenter runs at reduced capacity, all other datacenters start processing a larger share of work immediately.
- Because of continuous load balancing and higher availability, any delays that do occur tend to be shorter, and it is very rare for long delays to happen, so less capacity is needed to recover from long delays.

In a multi-homed system deployed in three datacenters with 20% total catch-up capacity, the total resource footprint is 170% of steady state. This is dramatically less than the 300% required in the failover design above.

Idle resources kept on standby for catch-up can sometimes be reused for other work. If the total resources for the system are small, it may be possible to acquire extra resources on demand. In extremely large systems, it is usually not possible to acquire resources on-demand because it is expensive and wasteful to run datacenters with low utilization. Additionally, in a disaster scenario, after one datacenter has been lost, many independent systems may be competing to acquire any available idle resources in alternate datacenters. If a system wants to guarantee that it will have spare capacity when it needs it, those resources must be pre-allocated.

6 Multi-homed Systems at Google

At Google, we have built several multi-homed systems to guarantee high availability and consistency even in the presence of datacenter level outages. This section summarizes how we solve some of the multi-homing challenges discussed in the previous section for these systems. The conference papers ([26], [3], [21]) have more details.

6.1 F1 / Spanner: Relational Database

F1 [26] is a fault-tolerant globally-distributed relational database built at Google to serve as the primary storage system for transactional data in AdWords and other advertising and commerce related applications. F1 is built on top of Spanner [15], which provides extremely scalable data storage, synchronous replication, and strong consistency and ordering properties. F1's core storage and transactional features are provided by Spanner. F1 adds a distributed SQL query engine, and additional database features for indexing, change recording and publishing, schema management, etc.

One of Spanner's primary purposes is to manage cross-datacenter replicated data. Data in a Spanner database is partitioned into *groups*, and each group typically has one replica *tablet* per datacenter, each storing the same data. Transactions within one group update a majority of replica tablets synchronously using the Paxos [23] algorithm. Multi-group transactions are supported using an additional two-phase commit protocol on top of Paxos.

Spanner has very strong consistency and timestamp semantics. Every transaction is assigned a commit timestamp, and these timestamps provide a global total ordering for commits. Spanner uses a novel mechanism to pick globally ordered timestamps in a scalable way using hardware clocks deployed in Google datacenters. Spanner uses these timestamps to provide multi-versioned consistent reads, including snapshot reads of current data. Spanner also provides a global safe timestamp, below which no future transaction can possibly commit. Reads at the global safe timestamp can normally run on any replica without blocking behind running transactions.

Spanner is a fully multi-homed system. For each group, one tablet is elected as the Paxos leader, and acts as the entry-point for all transactional activity for the group. A group is available for transactions if its leader and a majority quorum of replicas are available. If a leader tablet becomes unavailable or overloaded, an alternate tablet will automatically be elected as leader. During a full datacenter outage, all leaders will immediately move to alternate datacenters. During a partial outage, as machine capacity or performance degrades, leaders will gradually move to alternate datacenters. We commonly deploy Spanner instances with five replicas, so we can lose up to two datacenters and can still from a quorum of replicas and elect a leader tablet. (Five replicas is considered the minimum for high availability because during a full outage of one datacenter, availability must still be maintained during machine-level failures or transient failures in any remaining datacenter.)

The F1 system uses servers in front of Spanner that receive all F1 user traffic. The F1 servers are deployed in multiple datacenters and are mostly stateless. For writes, the F1 server must communicate with a Spanner leader replica. For reads, the F1 server can often communicate with a local Spanner replica. As an optimization, the system tries to shape traffic so that requests from a client, to F1, and to the Spanner leader replica all happen in one datacenter as often as possible. This is not required however, and after a failure anywhere in this stack, traffic will be transparently redirected to alternate datacenters to maintain high availability.

Transactions in F1 and Spanner require communication between replicas in multiple datacenters. We typically choose to distribute replicas widely across the US, which leads to commit latency of at least 50ms. Users coming from typical singly-homed databases are used to much lower latency. F1 users follow a set of best practices for application and interface design, emphasizing high throughput, parallelism and data clustering, to mitigate the effects of high transaction latency. We have found that by following these recommendations, applications on F1 can achieve similar performance to predecessor applications, and often have better long-tail performance because these approaches are inherently more scalable.

While F1 and Spanner are not stream processing systems, they provide good examples of the same multi-homing principles, and are often used as components inside other systems. Most of our stream processing systems require some form of global state for metadata, and sometimes for data, and this is usually stored in a Spanner or F1 database. These systems provide high availability reads and writes, with full durability and consistency, even during or after datacenter failure.

6.2 Photon: Joining Continuous Data Streams

Photon [3] is a geographically distributed system for joining multiple continuously flowing streams of data in real-time with high scalability and low latency. One of the key applications of Photon within Google's Advertising System is to join data streams such as web search queries and user clicks on ads. When a user issues a search query at google.com, Google serves ads to the user along with search results. The web server that serves the ad also sends information about this event to multiple logs-datacenters, where the data is stored persistently. The logged data includes information such as advertiser identifier, ad text, and online ad auction parameters. After receiving the results of the search query, the user may click on one of the ads. The click event is also logged and copied to multiple logs datacenters. Due to technical limitations on the size of the click URL, the click logs do not contain all the rich information in the original query log. Photon is responsible to join these streams of data arriving in multiple datacenters, and producing a joined log, which can then be consumed by other systems at Google to derive key business metrics, including billing and reporting for advertisers.

Given the business critical nature of the output joined logs, Photon is designed to tolerate infrastructure degradation and datacenter-level outages without any manual intervention. With the input logs available in multiple datacenters, Photon workers are able to run independently in each of these datacenters to join the same input event, but workers coordinate their output to guarantee that each input event is joined and outputted at-most-once. The critical state shared between the workers consists of the set of event_ids that have already been joined. This system state is stored in the *IdRegistry* (as illustrated in Figure 2), which is built using Paxos [23] and guarantees synchronous replication of the joined event_ids across the majority of its replicas. With this, Photon guarantees that there will be no duplicates in the joined output (at-most-once semantics) at any point in time, that most joinable events will be present in the output in real-time (near-exact semantics), and that exactly-once semantics are provided eventually.

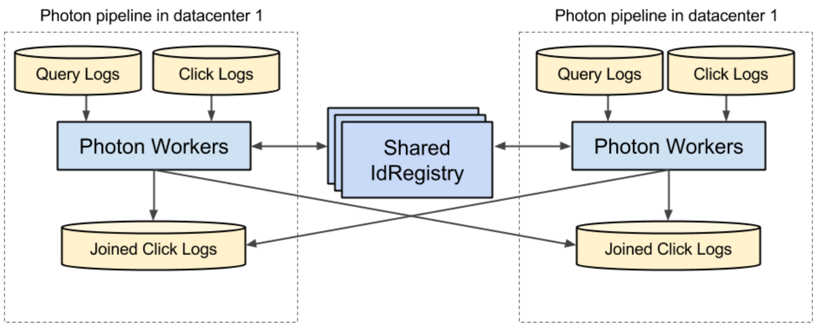

Fig. 2. Photon pipeline running independently in two datacenters with the IdRegistry storing the global state synchronously in multiple datacenters.

To ensure that an outage of one relatively large geographical region does not affect the IdRegistry, Photon places IdRegistry replicas in different geographical regions such that replicas from multiple geographical regions must agree before a transaction can be committed. The downside of requiring such isolation zones is that the throughput of the IdRegistry will be limited by network latency. Based on typical network statistics, the round-trip-time between different geographical regions (such as east and west coasts of the United States) can be over 100 milliseconds. This would limit the throughput of Paxos to less than 10 transactions per second, which is orders of magnitude fewer than our requirements—we need to process (both read and write) tens of thousands of events (i.e., key commits) per second. This was one of the biggest challenges with Photon, and by effectively using client and server side batching, and sharding of the IdRegistry (more details in the Photon [3] paper), Photon was able to scale to our needs.

Photon has been deployed in production at Google for several years, and it has proven to be significantly more reliable than our previous singly-homed and failover-based systems, handing all datacenter-level outages automatically, without loss in availability. For example, Figure 3 shows the numbers of joined events produced by a production pipeline in two separate datacenters over a

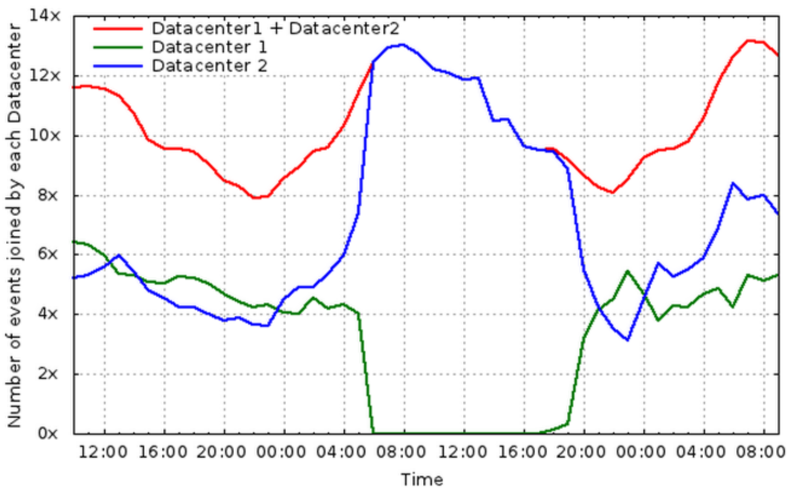

Fig. 3. Photon withstanding a real datacenter disaster

period of time. When both datacenters are healthy, each processes half of the total events. However, when one of the datacenters suffers from an outage, the other datacenter automatically starts handling the complete traffic without any manual intervention.

6.3 Mesa: Data Warehousing

Mesa [21] is a highly scalable analytic data warehousing system that stores petabytes of data, processes millions of row updates per second, and serves billions of queries per day. At Google, Mesa is used to store detailed information associated with each served ad, such as the targeting criteria, number of impressions and clicks, etc., which are recorded and processed in real time. This data is used extensively for different use cases, including reporting, internal auditing, analysis, billing, and forecasting.

Mesa is commonly used as the output storage system for stream processing systems. The streaming system consumes raw log events as input, applies business logic compute updates, and then aggregates these updates into multiple tables in the Mesa data warehouse. These Mesa tables store aggregates broken down by various metrics, so exactly-once event processing is necessary to provide accurate totals. Each input event can update aggregates in many tables, and end-users look at reports and graphs from multiple tables, expecting to see consistent totals.

Mesa is designed to provide consistent transactional updates in petabyte-scale tables with millions of updates per second, and uses multi-homing to provide high availability. This is possible because of Mesa's novel batch-update data model and transaction protocol. Mesa tables use multi-versioning, and each applied update batch gets a new version number. Update batches are written as deltas, and deltas are merged into the base in the background and as necessary at query time. A query at a particular version always sees consistent data at that version.

The large update volume in mesa makes it impractical to apply updates transactionally through Paxos-style commits. Instead, each update batch is copied asynchronously to multiple datacenters. A committer job waits until an update is copied to a quorum of datacenters, and then commits the update globally using a metadata-only Paxos commit. The version that is committed and queryable is always determined based on state in the central Paxos metadata database. This protocol scales to very large updates while providing full consistency and transparent handling of full or partial datacenter failures.

We use similar techniques in many of our streaming systems; Metadata indicating the state of work batches is synchronously committed in a global database, while the data is processed outside that transaction. For streaming systems, we often do most processing locally, then asynchronously replicate the results, and then commit. Like in Mesa, this technique minimizes synchronous global writes, enabling processing at very high scale and throughput, while still supporting multi-homing and consistent failover.

7 Related Work

There is a large body of early research on managing consistency for replicated data [10, 27]. The context since then has changed via the introduction of geo-replication. The wide-area latency in a geo-replicated environment introduces challenges not considered by the traditional approaches. Early geo-replication solutions experimented with weakening consistency guarantees to ensure high performance [14, 19, 4, 9]. More recently, however, there is wider interest in providing transactional guarantees to geo-replicated data. Spanner[15] is the first system to provide consistent geo-replicated transactions at scale.

Many stream processing systems have been built, but we are unaware of any published systems other than Photon[3] using geo-replication and multi-homing to provide high availability even in the presence of datacenter failures, and full consistency. Many systems have been built using relaxed consistency models that do not provide exactly-once semantics (Storm[7], Samza[6], Pulsar[8]). Systems that do provide exactly-once semantics either do not run at scale with fault tolerance (Aurora[1], TelegraphCQ[11], Niagara[13]) or do so within a single datacenter only (Dataflow[2], Spark streaming[28], Flink[5]).

8 Conclusion

As part of Google's advertising infrastructure, we have built and run several large-scale streaming systems, through multiple generations of system design, over many years. We have found that clients consider both consistency (exactly-once processing) and high availability to be critical properties. We have tried various approaches in various systems over time, and have learned that building natively multi-homed systems is the best solution in most cases.

Most of our systems began as singly-homed systems, and then some kind of multi-datacenter design was added on. When consistency requirements are low, running two singly homed pipelines is a simple solution that works well. When consistency is required, it becomes necessary to design failover procedures, which add significant complexity and cost.

Our experience has been that bolting failover onto previously singly-homed systems has not worked well. These systems end up being complex to build, have high maintenance overhead to run, and expose complexity to users. Instead, we started building systems with multi-homing designed in from the start, and found that to be a much better solution. Multi-homed systems run with better availability and lower cost, and result in a much simpler system overall.

The simplicity of a multi-homed system is particularly valuable for users. Without multi-homing, failover, recovery, and dealing with inconsistency are all application problems. With multi-homing, these hard problems are solved by the infrastructure, so the application developer gets high availability and consistency for free and can focus instead on building their application.

Based on our experiences, we now expect all of our systems, and any future systems we build, to have native multi-homing support as a key part of the design.

9 Acknowledgements

We would like to thank the teams inside Google who built and ran the systems we have described (both current and previous generations), including the Site Reliability Engineering teams that helped make high availability possible in these systems. We would like to thank Divyakant Agrawal for his help preparing this paper.

References

1. Abadi et al. Aurora: A new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, Aug. 2003.
2. Akidau et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment*, 8(12):1792–1803, 2015.
3. R. Ananthanarayanan et al. Photon: Fault-tolerant and Scalable joining of continuous data streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, (SIGMOD'2013), New York, NY, USA*, 2013.
4. Apache Cassandra, 2011. [Online; acc. 5-Oct-2011].
5. Apache Flink. `http://flink.apache.org`, 2014.
6. Apache Samza. `http://samza.apache.org`, 2014.
7. Apache Storm. `http://storm.apache.org`, 2013.

8. M. Astley et al. Pulsar: A resource-control architecture for time-critical service-oriented applications. *IBM Syst. J.*, 47(2):265–280, Apr. 2008.
9. P. Bailis and A. Ghodsi. Eventual Consistency today: Limitations, extensions, and beyond. *ACM Queue*, 11(3):20:20–20:32, Mar. 2013.
10. P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.
11. Chandrasekaran et al. TelegraphCQ: Continuous Dataflow Processing. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 668–668, New York, NY, USA, 2003. ACM.
12. F. Chang et al. Bigtable: A Distributed Storage System for Structured Data. In *7th Symposium on Operating Systems Design and Implementation (OSDI'2006), November 6-8, Seattle, WA, USA*, pages 205–218, 2006.
13. Chen et al. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, SIGMOD '00, New York, NY, USA, 2000. ACM.
14. Cooper et al. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, Aug. 2008.
15. J. C. Corbett et al. Spanner: Google's Globally-Distributed Database. In *10th USENIX Symposium on Operating Systems Design and Implementation, (OSDI'2012), Hollywood, CA, USA, October 8-10, 2012*, pages 261–264, 2012.
16. G. Cormode and M. N. Garofalakis. Approximate continuous querying over distributed streams. *ACM Trans. Database Syst.*, 33(2), 2008.
17. G. Cormode, S. Muthukrishnan, K. Yi, and Q. Zhang. Continuous sampling from distributed streams. *J. ACM*, 59(2):10, 2012.
18. J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *6th USENIX Symposium on Operating System Design and Implementation (OSDI'2004), San Francisco, California, USA*, pages 137–150, 2004.
19. G. DeCandia et al. Dynamo: Amazon's highly available key-value store. In *Proc. 21st ACM Symp. Operating Systems Principles*, pages 205–220, 2007.
20. P. Flajolet and G. N. Martin. Probabilistic counting algorithms for data base applications. *J. Comput. Syst. Sci.*, 31(2):182–209, 1985.
21. A. Gupta et al. Mesa: Geo-Replicated, Near Real-Time, Scalable Data Warehousing. *PVLDB*, 7(12):1259–1270, 2014.
22. L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, 1978.
23. L. Lamport. The Part-Time Parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.
24. A. Metwally, D. Agrawal, and A. El Abbadi. An integrated efficient solution for computing frequent and top-k elements in data streams. *ACM Trans. Database Syst.*, 31(3):1095–1133, 2006.
25. N. Shrivastava et al. Medians and beyond: new aggregation techniques for sensor networks. In *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems, SenSys'2004, Baltimore, MD, USA*, 2004.
26. J. Shute et al. F1: A Distributed SQL Database that Scales. *PVLDB*, 6(11):1068–1079, 2013.
27. G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan-Kaufman Publishers, 2002.
28. Zaharia et al. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 423–438. ACM, 2013.