Owl: Scale and Flexibility in Distribution of Hot Content

Jason Flinn, Xianzheng Dou, Arushi Aggarwal, Alex Boyko,
Francois Richard, Eric Sun, Wendy Tobagus, Nick Wolchko, Fang Zhou
Meta

Abstract

Owl provides high-fanout distribution of large data objects to hosts in Meta's private cloud. Owl combines a decentralized data plane based on ephemeral peer-to-peer distribution trees with a centralized control plane in which tracker services maintain detailed metadata about peers, their cache state, and ongoing downloads. In Owl, peer nodes are simple state machines and centralized trackers decide from where each peer should fetch data, how they should retry on failure, and which data they should cache and evict. Owl trackers provide a highly-flexible and configurable policy interface that customizes and optimizes behavior for widely-varying distribution use cases. In contrast to prior assumptions about peer-to-peer distribution, Owl shows that centralizing the control plan is not a barrier to scalability: Owl distributes over 800 petabytes of data per day to millions of client processes. Owl improves download speeds by a factor of 2–3 over both BitTorrent and a prior decentralized static distribution tree used at Meta, while supporting 106 use cases that collectively employ 55 different distribution policies.

1 Introduction

Within Meta's private cloud, efficient distribution of large, hot content to end hosts is an increasingly important requirement. Three dimensions express the scope of the task: (1) scale: the same content may be read by anywhere from a handful of clients to millions of processes running in data centers around the globe, (2) size: objects to be distributed range from 1 MB to a few terabytes, and (3) hotness: all clients may read an object within a few seconds of each other, or their reads may be spread over hours. At Meta, executables, code artifacts, AI models, and search indexes are content types commonly distributed within this scope.

Distribution requirements are exacting. First, content distribution must be *fast*: the predictive value of AI models decreases over time, and slow executable delivery increases downtime and delays deploying fixes. We expect to provide data at a rate bounded by either the available network bandwidth of the reading host or by the available write bandwidth of its storage media.

Second, content distribution must be *efficient*. One dimension of efficiency is scalability, i.e., the number of clients that can have their distribution needs met by a given number of servers. Another dimension is network usage, which we measure both in terms of bytes transmitted and communication locality (e.g., an in-rack data transfer is less costly than a cross-region transfer) . A final dimension of efficiency is resource usage on client machines; e.g., CPU cycles, memory, and disk I/O. Not only should we use as few resources as possible, but we should also adjust for their relative importance on different clients; e.g., some services are memory-constrained, while others are CPU-constrained or cannot afford to write to disk.

Finally, content distribution must be *reliable*. Reliability is measured as the percentage of download requests that the distribution system satisfies within a latency SLA. Operational ease-of-management is an oft-overlooked prerequisite for high reliability. In a production environment, workloads change, dependent service and infrastructure may have partial outages, and performance faults in which a dependency doesn't meet its own SLA are not uncommon. In order to maintain a high SLA for distribution, engineers need to be alerted quickly about such events, and they need a clear picture of operational health for each client type. Finally, they need simple knobs that adjust behavior when reliability, speed, or efficiency starts to degrade in order to restore operational health quickly.

Prior to our work, Meta used at least three different systems for large content distribution. No prior solution met all of the above requirements. We identified two root causes: (1) no prior system struck the correct balance between decentralization and centralization, and (2) no prior system was sufficiently flexible to meet all of the requirements of the many different types of services at Meta that require content distribution.

Meta previously implemented highly-centralized distribution via hierarchical caching, in which clients download content from first-level caches on remote hosts. These caches, in turn, handle cache misses by reading from other caches, with the final layer of the hierarchy being a distributed storage system. Hierarchical caching is inefficient for hot content distribution, and it is difficult to scale. Meta needed dedicated hosts in great quantity to implement the cache hierarchy. The number of hosts increased to keep pace with growth in workloads from services consuming the data and with growth in the number of reading clients. Load spikes caused by hot content were a continual problem: strict quotas were necessary to protect the centralized caches. However, readers of hot content were frequently throttled because they exceeded their quotas. In general, provisioning for transient spikes caused by hot content and setting quotas appropriately was quite challenging.

Meta also used two highly-decentralized systems: a location-aware BitTorrent [7] implementation and a static peer-to-peer distribution tree based on consistent hashing, which we will refer to as StaticTree. In both cases, a peer is any process that wishes to download data, and there are millions of such processes at Meta. The decentralized systems scaled much better than hierarchical caching, but they brought their own problems. First, because each peer made distribution decisions based on local information, resource efficiency and speed could be poor; e.g., with each peer making independent caching decisions, the collection of peers could retain either more or less copies of a data object than necessary. Perhaps more importantly, these decentralized solutions were difficult to operate. Engineers could not get a clear picture of health and status without aggregating data from large numbers of peers. Each peer had a different and limited view of the state of distribution, so it was often hard to tell whether or not a collection of peers was making good decisions. In general, it was very hard to reason about system-wide correctness or efficiency.

In summary, decentralized systems were inefficient and difficult to operate, while centralized systems scaled poorly. As a result, we chose to create a new, split design with a decentralized data plane and a centralized control plane. The decentralized data plane streams data from sources to clients via a distribution tree. However, its trees are *ephemeral*, i.e., each tree tracks a single data chunk, and each edge in a tree persists only while the chunk is being transferred from a source to a peer.

The design realizes a mechanism-policy split. Peers are simple and provide the mechanism for caching and transferring data chunks. The centralized control plane makes all detailed policy decisions about distribution, e.g., from where peers should get each chunk of content, when and how they should cache content, and how they should retry failed downloads. The control plane is implemented by a small set of *trackers* [1]. Trackers have a complete picture of the distribution state; e.g., which data each peer is downloading, where these peers are located, and which chunks are in each peer's cache. Detailed state enables trackers to make highly-optimized decisions about data placement and distribution that minimize the use of expensive network links and maximize cache hit rate. Centralizing the control plane has also made distribution easy to operate and debug: engineers can understand which decisions led to low availability, high latency, or poor hit rate because these decisions are made by a tracker with a consistent view of distribution state.

When workloads scale beyond the capacity of a single tracker, the detailed state is sharded across several cooperating trackers, each managing a distinct set of peers. Trackers exchange lower-fidelity views of their individual state with other trackers. Thus, each tracker has a fine-grained view of the state it manages and a coarse-grained view of the entire state. Trackers use the coarse-grained view to delegate decisions to other trackers when using peers that those trackers manage.

The second major problem faced by prior distribution systems was a lack of flexibility. At Meta, clients have vastly different resources to spare for distribution; e.g., some clients can dedicate gigabytes of memory or disk for peer-to-peer caching, while others have no resources to spare. Client have very different access patterns and scale. Finally, the objectives for distribution can differ: some clients need low latency, while others wish to reduce the load on external storage to avoid throttling or excess quota requests. The variety in client needs was one reason Meta needed many different distribution solutions; each solution was customized for a small set of use cases. To unify the disparate distribution solutions, we could not simply provide a one-size fits all solution because that would regress many clients on their key metrics.

We therefore chose to make customization a first-class design priority. Trackers implement modular interfaces for specifying different *policies* for caching and fetching data. Further, each policy is itself configurable to allow for different tradeoffs across client types and responses to changing workloads. We use trace-driven emulation to search through the space of possible customizations and find the best policies and configurations for each observed workload.

This paper describes our solution, Owl, a highly-customizable data distribution system with a centralized control plane and a decentralized data plane. Owl has been in production use at Meta for almost 2 years. Owl has scaled out rapidly (production traffic increased by almost 200x in 2021). Currently, Owl has over 10 million unique clients (binaries concurrently using the Owl library), and it downloads over 800 petabytes of data per day. Owl supports 106 unique types of clients and has customized policies for 55 of these. In production, Owl improved download latency over prior systems by a factor of 2–3 for our most important use cases, while requiring only a fraction of the resources needed by prior centralized solutions.

In summary, this paper makes the following contributions:

1. It shows that a centralized control plane need not be a barrier to scalability in peer-to-peer distribution.

2. It shows that tracker sharding and delegation retain the benefit of fine-grained management even when load grows beyond the capacity of a single tracker.

3. It shows that first-class support for flexible distribution

[1] borrowing terminology from BitTorrent

Client API function and arguments	**Description**
`read_blob (object, offset, length, deadline, integrityChecker, decryptor)`	Fetches all or part of an object to memory.
`read_blob_to_file (object, fd, offset, length, deadline, integrityChecker, decryptor)`	Fetches all or part of an object to a file.
`provide_file (object, fd, length)`	Allows a file to be distributed ephemerally.
`evict_file (fd)`	Evicts a file from the peer cache.

Table 1: **Owl client API**

and caching policies can provide substantial gains in efficiency and latency, especially when combined with tools that automatically search the space of possible policies for optimizations.

2 Design and Implementation

Owl has two basic components: *peers*, libraries linked into every binary that uses Owl to download data, and *trackers*, dedicated Owl services that manage the control plane for a group of peers. A physical host often has several Owl peers due to container stacking and use of Owl by the Twine container infrastructure [16]. Each tracker manages many peers: over 10 million Owl peers are currently managed by 112 trackers. Additionally, Owl has approximately 800 *superpeers*, dedicated services running the Owl library that provide extra caching or perform specialized tasks.

2.1 Peers

Owl peers provide a simple API for downloading data, shown in Table 1. Client processes call `read_blob` to fetch content from a source object, specifying a range of data to read. The object name encodes an external storage source and a unique identifier for the object within the external storage namespace. Owl currently supports 3 types of external storage. The caller can optionally specify a deadline and classes that check data integrity or decrypt provided data (discussed in Section 2.9). `read_blob` returns a reference-counted memory buffer, while `read_blob_to_file` writes the content to a file. The `provide_file` function allows peers to provide ephemeral content, as discussed in Section 2.11, and `evict_file` lets clients manage disk caches shared with Owl, as discussed in Section 2.7.

Owl peers cache data in memory and on disk. These caches may be shared with the client binary if the client does not modify downloaded data. Owl uses the caches to serve content requests from other peers. Owl policies usually prefer to fetch data from a peer rather than from an external data source, so most requests are satisfied by peer-to-peer distribution.

In the design of Owl, a key principle is that peers should be as simple as possible. This is achieved via a mechanism-policy split, where peers provide the mechanism to perform simple actions such as downloading a chunk of content from a single source, caching or evicting a chunk in memory or on disk, or providing cache data in response to a request from another peer. When downloading content, peers ask trackers to decide from where they should fetch content, how they should retry failed downloads, and even which chunks they should cache locally.

This design principle has been invaluable for operational simplicity. At Meta, the Owl team can control the deployment of its own service (i.e., trackers and superpeers); however, Owl peers are linked with client binaries and so deploy according to different schedules controlled by many other teams. The Owl team deploys code changes to trackers daily, and the team can change configuration values on trackers within seconds if necessary. In contrast, peer code changes can take months to fully deploy. By keeping peers as simple as possible, the team minimizes the need to change a widely-deployed and hard to modify part of the system.

Each peer is associated with a *bucket*, which uniquely identifies the type of the client binary with which the library is linked. The bucket provides a way to customize Owl behavior for each type of client and it lets us monitor usage, performance, and reliability for each Owl customer individually. Currently, Owl supports production traffic for 106 buckets.

2.2 Trackers

A tracker manages download state for a set of peers. Typically, peers and trackers are grouped by region (a region is several co-located data centers), with 3–4 trackers per region providing scale and redundancy. Trackers are homogeneous and multi-tenant. In general, each tracker supports all Owl buckets, and the association between peers and trackers in a region is random. However, Owl uses a separate set of trackers in each region for binary distribution to provide strict performance isolation for this sensitive workload.

Trackers associate data and peers. Downloaded objects are divided into chunks; chunk size varies by bucket with 50 MB being the most common size. For each chunk, tracker metadata specifies which peers are caching the chunk and which are downloading it. Tracker metadata also specifies the source of each peer's download (e.g., an external source or another peer). For each peer, the tracker metadata specifies the peer's location (host, rack, region, etc.) and its cache state (the chunks in the cache, last access time, and so on). In contrast to highly-decentralized systems, Owl trackers can maintain such detailed state because trackers make all major decisions about caching and downloading chunks on behalf of peers.

As our evaluation shows, a single Owl tracker can scale to handle 1.5–2.4 TB/s of distribution traffic, depending on assumptions about cache hit rate for download requests. To achieve this scalability, we have used careful, but mostly standard, engineering practices. Trackers are implemented in C++ and use common abstractions (coroutines, reader-writer locks, and standard library containers). Trackers maintain geographically-sorted indexes to order peers and the chunks they cache by location; these indexes allow trackers to efficiently find the nearest peers caching a particular chunk of data. Geographically-sorted indexes are used frequently by location-aware selection polices. Trackers store all metadata in memory, and they rebuild their state quickly when restarted.

Peers associate with one tracker. Each peer picks a random instance from the set of available trackers and registers by sending an RPC. Peers register with a new random tracker if their association with the current tracker fails. Section 2.8 describes how peers are sharded across multiple trackers.

2.3 Superpeers

Superpeers are tasks running the Owl peer library as a standalone process (without any client). Superpeers sometimes provide specialized functionality. For example, some external storage systems use mountpoints that are not available on most hosts, so we access this storage only via superpeers that have been configured with the necessary mountpoint. To read external storage, the tracker directs such a superpeer to fetch and cache a data chunk, and it directs a reading peer to get the data from the superpeer.

Some Owl buckets need more peer resources than their clients can provide. For example, some clients are extremely memory and disk constrained and yet also require a high cache hit rate to reduce load on external storage. Superpeers can use all the resources of their hosts for caching, and so Owl uses superpeer caches to supplement peer caches for such buckets.

When used in this manner, a collection of superpeers can be viewed as a hierarchical caching layer. It is possible to craft Owl selection policies that direct all requests to superpeers and bypass fetching from other peers. Early in the project, we created one such policy to support an AI bucket that could spare no memory or disk for peer caching. However, we soon found that shared caching, discussed in Section 2.7, allowed Owl to temporarily access data buffers in use by the application to provide a decent peer-to-peer cache hit rate. Superpeers are still valuable because their additional caching resources improve the total Owl cache hit rate from the base peer-to-peer rate up to the target needed by the AI team. Currently, the Owl team discourages superpeer-only policies because there are several existing systems at Meta that provide excellent standalone caching solutions.

At the other end of the spectrum, it is also possible to craft Owl selection policies that do not use superpeers at all. For buckets with very large working sets and large numbers of clients, the additional cache resources of superpeers make little difference in overall cache hit rate. In practice, though, Owl selection policies for these types of buckets still use superpeers as a last level of retry if a direct fetch from storage by the peer fails. We have found this to be useful in handling rare corner cases such as particular peers being in a bad state where they cannot fetch from external storage. Because peers run on heterogeneous hosts not owned by the Owl team, they can be less stable than superpeers. The superpeer layer thus still plays a role in improving overall data availability.

Superpeers have occasionally been quite useful in mitigating production issues that lead to a poor cache hit rate. In such scenarios, we have quickly stood up a large number of superpeers in a region to provide temporary caching that restores the desired hit rate until we are able to deploy a fix for the underlying problem.

We originally implemented superpeers as a sharded service built on a standard caching library [3]. This approach proved to be insufficiently flexible; it was difficult to customize superpeer cache behavior for each bucket. Later, we rewrote superpeers to use the Owl peer library, which let us customize superpeers via tracker policies and which also provided the simplicity of code reuse for peers and superpeers. From the point of view of a peer, there is no distinction between fetching a data chunk from a superpeer or another peer.

2.4 Tracker-Peer Communication

Peers register with a tracker by sending an RPC with their bucket and location. Trackers customize behavior by bucket; e.g., tracker configuration parameters assign specific download and caching policies to each bucket. When registering, peers select trackers randomly within a geographic scope, and the association between a tracker and peer persists until a peer terminates or cannot communicate with the tracker. On failure to communicate with a tracker, peers re-register with a randomly-chosen tracker.

To download data, an Owl peer first makes an RPC to the tracker specifying the object to be downloaded and the range of data to read. The tracker returns the chunk size (determined by the bucket configuration). It initializes a state machine to track the download of each chunk that has data in the specified range. If the download later fails or times out, the tracker cancels all per-chunk state machines for the download. Otherwise, each chunk is handled independently.

Peers download chunks in parallel. Download concurrency is limited by the per-bucket configuration, which specifies both the maximum number of chunks each peer can download in parallel and a maximum number of chunks that can be read

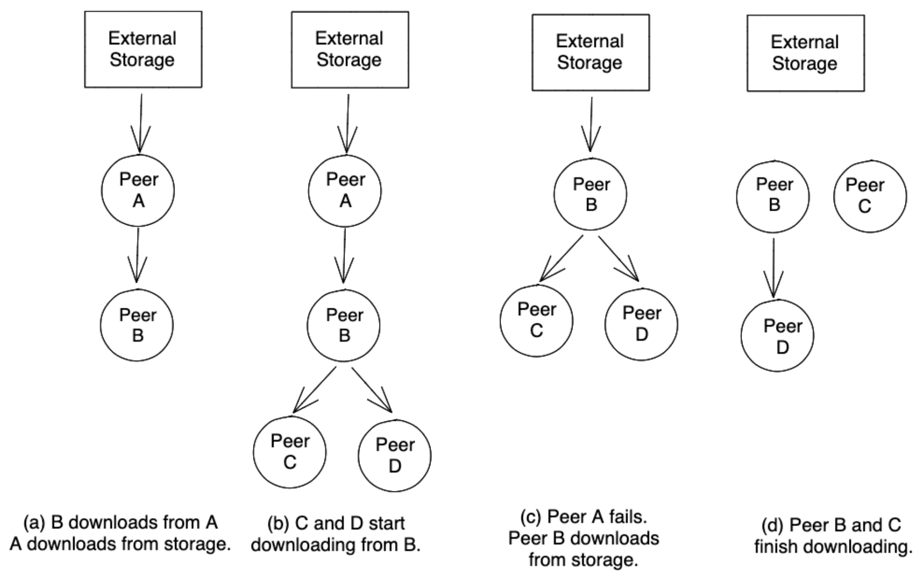

Figure 1: **Ephemeral distribution tree.** This figure shows how a tracker uses a per-chunk ephemeral distribution tree to track which peers cache a chunk and which are downloading the chunk.

in parallel for each individual `read_blob` request.

For each chunk, the peer first checks if the chunk is in its cache already. If so, it sends an RPC to the tracker, which terminates the per-chunk download state machine and updates the access time in the peer metadata for LRU and similar eviction policies. Otherwise, the peer sends a `getSource` RPC to the tracker that asks how it should get the data. The tracker can respond with a peer from which data can be obtained. Alternatively, the tracker may specify an external data source from which the peer should fetch content directly. The `getSource` response specifies whether the peer should cache the downloaded chunks and lists chunks that should be evicted from the peer cache to make room. The tracker updates its peer and chunk metadata, along with the per-chunk download state machine to reflect its decision.

The peer next attempts to obtain the data in the manner specified by the tracker, and it informs the tracker of the result. On success, the tracker terminates the state machine. On failure, the tracker makes a new decision based on its current metadata. Based on per-bucket policies and the failure type, the tracker may specify a new source (or possibly retry the same source in rare cases), or it may tell the peer to give up (e.g., because a maximum number of retries has been exceeded or it believes the chunk is not available from any source).

Prior systems such as BitTorrent [7] provide a list of candidate sources to a peer and let peers handle retries transparently. Owl's approach of involving trackers in retry decisions has several advantages. First, trackers can pick a new source based on the latest state about which peers cache the chunk and current peer load. In contrast, the peers included in BitTorrent's initial list can be stale when retries are needed. Second, Owl trackers maintain very detailed state about which peers are fetching from others. This allows trackers to enforce precise caps on the maximum number of inflows and outflows per peer, and it allows trackers to make more informed selection decisions. Finally, this detailed state gives operators a complete picture of Owl download state, making it easier to determine why downloads may be slow or failing.

The peer simply follows the tracker's instructions at each step. On retry, if a prior step returned partial content before failing, the peer resumes fetching from a new source after the last byte it received (so failures do not lead to excess data being transmitted).

If a chunk download fails (because the tracker tells the peer to give up) or the download time exceeds a deadline specified by the client, the peer cancels all remaining chunk downloads and fails the `read_blob` request. Otherwise, the peer returns the requested data either via a memory buffer or by writing to a specified disk file. The tracker also sets a timeout for each chunk download; it terminates a download and cleans up download state if the peer does not respond within this time.

2.5 Ephemeral Distribution Trees

The *ephemeral distribution tree* is the core abstraction used by the tracker to manage per-chunk download state. The root of a tree is an external data source or a peer that caches the chunk. Directed edges indicate which peers are actively downloading the chunk from others; e.g., in Figure 1(a), peer A is downloading the chunk from external storage, and peer B is downloading the chunk from peer A. While prior distribution systems have commonly used trees to efficiently distribute data, Owl's trees are particularly ephemeral in that each chunk of data has its own forest of trees, and nodes remain in a tree only while they are downloading a particular chunk or providing a chunk to another peer.

In the data plane, chunks are streamed from the root to leaves; i.e., bytes along each edge are sent in order, followed by a per-chunk checksum used to verify integrity. Each peer forwards data to its children as soon as it receives new bytes. With large chunks, this design means that tree depth does not strongly affect latency. Leaf-nodes see only the first-byte latency of additional communication hops, which is often quite small within a data center or region. In Figure 1(b), when peers C and D request the chunk; the tracker tells them to get it from B, which is still receiving data. Peer B first sends its cached bytes and then forwards additional chunk bytes as it receives them.

When a peer reports a failure fetching data, the tracker removes the edge connecting the peer to its parent. If the tracker chooses a new source, it creates an edge from the peer to that source. Thus, the entire subtree rooted at the peer reporting the failure is moved to a have a new parent in the tree. When choosing a new peer, the tracker avoids creating download cycles; i.e., it will not designate a descendent of a peer as a new source for that peer. Tree repair minimally impacts downstream nodes because Owl resumes a new download after the last byte fetched from the previous attempt. In Figure 1(c), peer A fails, and the tracker tells peer B to fetch the remaining bytes from external storage. Peers C and D are oblivious to this change since they continue to download from B.

When a peer reports a successful download, the edge connecting it to its parent is removed. The tracker adds the peer to the list of nodes that have the chunk fully cached if the tracker asked the peer to retain the chunk. Since chunks may be cached at multiple peers, the download state for a chunk is a forest of ephemeral distribution trees rooted at multiple such peers and/or the external data source. In Figure 1(d), peers B and C have downloaded and cached the chunk, while peer D is still downloading bytes from B. Thus, we have two ephemeral distribution trees in the forest; a new peer that requests the chunk may be directed to any of these peers or to external storage, depending on the selection policy for the bucket.

At first glance, it might seem surprising that Owl often prefers to download chunks from peers that have partially downloaded a chunk in preference to peers that have the chunk fully cached. However, selecting a peer that has partially downloaded a chunk has little latency cost. The peer immediately starts streaming out the bytes it has already downloaded and sends remaining bytes out as soon as they arrive. Network locality and quick scale-out of hot contents are bigger concerns in practice. For instance, many peers in a rack often request a chunk at the same time. Most Owl policies are location-aware and build a tree so that a single peer downloads data from outside the rack and other peers in the same rack get the chunk from that peer or one of its children. Similarly, if many peers in a data center request a chunk at the same time, typically only one peer fetches the chunk from outside the data center. Allowing peers to fetch from other peers that are still downloading the chunk is essential to achieving network locality for hot content.

2.6 Selection Policies

Each bucket has a selection policy that the tracker executes on each `getSource` request. The selection policy considers the result of all prior attempts by a peer to fetch a chunk, as well as per-chunk state that includes the set of caching peers and ephemeral distribution trees. The result of the selection policy often directs a peer to fetch the chunk from another peer or an external data source; these decisions add a new edge to an ephemeral distribution tree. The policy is implemented as a class inheriting from an abstract interface; each policy class has a considerable number of parameters that can be further customized via configuration [15].

A selection policy may use a superpeer to assist in the download. The tracker directs the superpeer to fetch the chunk from an external source, and it directs the requesting peer to get the chunk from the superpeer. This creates a 2-edge distribution tree. Usually, the tracker will have the superpeer cache the chunk so other peers can fetch the same chunk without reading from external storage; this is especially useful when many chunk requests arrive within a short time window.

The *location-aware* policy is the default selection policy. This policy selects the nearest peer that caches or is downloading the chunk, subject to per-peer constraints on maximum fanout and bandwidth usage. Distance is determined by network topology; peers on the same host are preferred over peers in the same rack, which are, in turn, preferred over peers in the same network cluster, etc. To make location-aware selections quickly, the tracker maintains a topological sort over all peers caching or downloading a chunk. A selection policy also specifies the number and type of retries. By default, the location-aware policy tries up to 5 peers, then tries to fetch the chunk via a superpeer, then tries to fetch the chunk directly from a source before giving up. The policy also has unique handling for specific errors, such as external source throttling.

Another common policy is the *hot-cold* policy, which refines the location-aware policy by using superpeers for hot data. If no peer can provide a chunk from its cache, this policy reads data from an external source via superpeers if the chunk is hot or directly from the source if the chunk is cold. Hotness is determined by examining the number of chunk reads within a recent time window. The policy improves hit rate in superpeer caches for buckets that have a mix of hot and cold content.

Other policies implement load balancing; e.g., spreading

```
1  Decision selectSourceForData(
2      const ChunkMetadata& MD,
3      std::shared_ptr<PeerMetadata> requester,
4      const ChunkStatus& stat,
5      const ShardedChunksMap& shardedChunks,
6      const DownloadContext& context) override {
7
8    if (hasDirectFetchFailed(stat)) {
9      return Decision{GIVE_UP, nullptr};
10   }
11
12   if (cannotFindSuperpeer() ||
13       noMoreSuperpeerAttempts(stat) ||
14       noMoreAttempts(stat)) {
15     return Decision{DIRECT_FETCH, nullptr};
16   }
17
18   if (noMorePeerAttempts(stat)) {
19     return Decision{SUPERPEER_FETCH, nullptr};
20   }
21
22   peer = selectPeer(MD, requester, stat, context);
23   if (peer) {
24     return Decision{PEER_FETCH, peer};
25   }
26
27   delegation = findDelegation(shardedChunks);
28   if (delegation) {
29     return Decision{DELEGATED_FETCH, delegation)};
30   }
31
32   return Decision{SUPERPEER_FETCH, nullptr};
33 }
```

Figure 2: Pseudocode: Location-Aware Selection Policy

downloads from sources evenly. Still others always fetch via superpeers, select random peers, and direct whether and how chunks should be fetched from out-of-region peers.

To illustrate how policies are written, Figure 2 shows pseudocode for Owl's location-aware selection policy. Each policy is implemented by overriding a C++ base class; in this case we show the `selectSourceForData` method, which is used to determine how and from where a peer should fetch data on each chunk download attempt. The method's inputs are: chunk metadata that includes a topologically sorted index of all peers and superpeers caching the chunk, metadata describing the peer requesting the data that incudes its location info, a status object containing all prior attempts to fetch the chunk for this download and their results, a list of other trackers that have the chunk available for delegation, and bucket-specific context about the chunk.

Policy implementations are usually a series of simple rules. The location-aware policy first calls a helper function (line 22) to select the nearest peer or superpeer caching or downloading the chunk, as long as such a peer is healthy (no recent failures reported) and would not exceed limits on number of downloads, network bandwidth, etc. The helper function considers past attempts and only tries each source once.

If the tracker has no more locally-managed peers or superpeers caching the chunk, it tries to find a delegation for the chunk from a peer tracker (line 27). If this fails, the policy asks a superpeer to fetch the chunk from an external source, cache it, and provide it to the requester (line 32).

The policy has configurable limits on the number of peer and superpeer attempts. If there are no more peer attempts allowed, the next retry asks a superpeer to fetch the chunk from an external source (lines 18–19). If there are no more superpeer attempts left or the policy has attempted to find a free superpeer and failed, the peer is asked to fetch the data from the external source directly (lines 12–15). If this direct fetch fails, the policy gives up (lines 8–9).

2.7 Caching policies

Per-bucket caching polices determine how peers cache data. Peers may cache data in memory or on disk, with some buckets using both types of cache. Cache size is configurable; the default memory cache size is 1 GB but size varies widely across buckets, depending on memory constraints and desired cache hit rates.

Some buckets use a *shared* peer cache, in which a single copy of data is shared read-only between the client application and Owl distribution. For in-memory caching, Owl returns a reference-counted buffer from `read_blob`. The buffer remains in the cache until the client releases the reference. For example, one memory-constrained client type reads AI models using a shared cache. While the client has no memory to spare, it retains data read for several seconds while it transforms the model chunk into a different format. By sharing the buffer with the client, Owl can satisfy many peer-to-peer download requests during this time. This sharing is essentially free because the data would reside in memory for the transformation anyway. This particular bucket needed a good hit rate to avoid overloading its external storage. Shared caching got us most of the way there, and we used superpeer cache capacity to further improve the hit rate to meet the bucket's requirement.

Buckets with disk caching often use shared caching. Downloaded objects are written to files with Owl retaining an open file handle so that it can serve cached file content to peers. The client controls when files are garbage collected by calling `evict_file` in Table 1. Owl also provides an interface that watches downloaded files and calls `evict_file` on the client's behalf if the file is deleted. In lieu of controlling eviction explicitly, some clients provide a *TTL* (time-to-live) that specifies how long downloaded data should be cached before eviction.

For private (non-shared) caching, Owl trackers manage peer caches. On each `getSource` request, the caching policy determines whether the peer should cache the requested chunk and which chunks to evict to make room in the cache. Peers specify their current cache state when registering with a new tracker so management persists across tracker failures.

The default caching policy is LRU (least recently used). Another popular policy, used for shared caching, never evicts chunks because the eviction is done explicitly by each peer.

Many clients that need good peer-to-peer cache hit rates use a *least rare* policy that prefers to evict chunks cached on more peers over chunks cached on fewer peers. A hybrid policy uses least-rare eviction for hot chunks and LRU eviction for cold chunks. Owl also supports random chunk eviction, which often has good properties for hot data [18].

2.8 Tracker sharding

For the first year of operation, Owl used a single tracker per region, with hot spares providing primary-backup fault tolerance. The simplicity of a single tracker allowed us to start serving production traffic 3 months after the start of the project. However, we knew that our workload would eventually exceed the capacity of a single tracker. Thus, we added the capability to shard peers across multiple trackers.

With sharding, trackers have equivalent responsibilities. A sharded tracker maintains the complete peer state for a given set of peers, but per-chunk and per-download state is split across the shards. Peers and superpeers register with random trackers.

Sharded trackers periodically exchange the set of chunks cached by at least one peer or superpeer that they manage. Trackers normally send incremental updates once a second with additions to and removals from this set. However, a receiving tracker may request a full snapshot when needed; e.g., because it just restarted or it missed an incremental update. Thus, each tracker has a coarse-grained and slightly stale view of the global distribution state that maps chunks to trackers rather than to specific peers.

Selection policies can decide to fetch a chunk from another sharded tracker; typically, this happens when the chunk is not cached on any peer managed by the local tracker and another tracker has reported that it has the chunk. The tracker running the selection policy sends a *delegation* request to the other tracker. In turn, that tracker selects and returns a peer caching or downloading the chunk. The delegation request fails if no such peer exists.

On successful delegation, each tracker updates state for the peer it manages. The `getSource` response simply specifies the endpoint of the delegated peer, so peers are oblivious to delegation. When the downloading peer reports success or failure, its tracker forwards the report to the delegating tracker and both trackers update their individual state accordingly.

On receiving a successful delegation response, a tracker starts a new ephemeral distribution tree. The root of a tree is a *delegated peer*, which indicates that the peer is managed by another tracker. The tracker grows the tree as other peers request the chunk, since selection policies commonly prefer to fetch from a locally-managed peer over a delegated one.

The ephemeral distribution tree for a chunk is now partitioned across multiple trackers with a node in the tree of one tracker serving as the root of a subtree in another tracker. In order to prevent cycles in this partitioned tree, a tracker will not provide any peer in a tree rooted at a delegated peer in response to a delegation request.

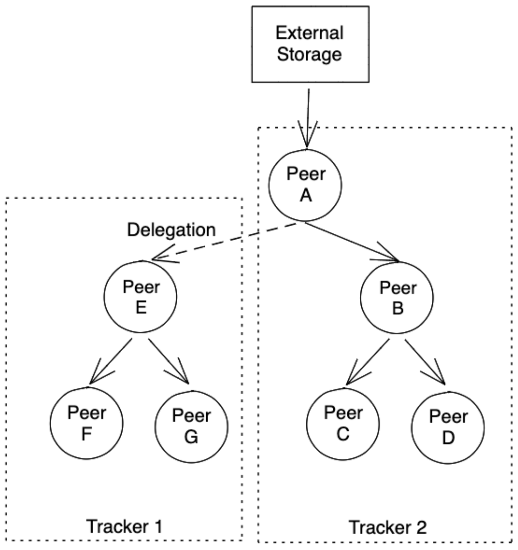

Figure 3: **Delegation with 2 sharded trackers.** Peer E fetches a chunk from a peer managed by another tracker to reduce load on external storage.

Figure 3 shows an ephemeral distribution tree sharded between 2 trackers. Tracker 2 initially receives a `getSource` request from peer A and instructs peer A to read the chunk from external storage. At this point, tracker 2 starts advertising that it has the chunk to other sharded trackers. Next, tracker 1 receives a `getSource` request from peer E. It does not have the chunk on any of its peers, but it knows that tracker 2 has advertised the chunk. Tracker 1 sends a delegation request to tracker 2, which selects and returns peer A. Tracker 1 tells peer E to fetch the chunk from peer A. When tracker 1 receives subsequent `getSource` requests from peers F and G, the bucket's selection policy prefers locally-managed peers, and so these peers are directed to fetch from peer E. As this example shows, delegation improves cache hit rate for sharded trackers. Without delegation, both peers A and E would fetch from external storage. With delegation, there is only a single fetch by peer A, which achieves the same overall cache hit rate that would have been achieved without sharding.

Some data sources accessed by Owl are regional. In these cases, when an out-of-region peer requests a chunk that is not cached by another peer, selection policies use delegation to ask an in-region tracker to have one of its superpeers read the chunk. The requesting peer is directed to that superpeer for the data. Selection policies consider cross-region latency to find the closest location from which to read data.

2.9 Security and integrity

All communication between Owl components in encrypted, and all RPCs are checked against access control lists. Many data sources read by Owl encrypt data at rest, so chunks in Owl caches are often encrypted. Owl clients can provide decryption functions to read decrypted data. For shared caching, Owl must cache and share unencrypted data with clients (because that is how they consume the data). In this case, Owl decrypts each chunk when writing it to the cache and re-encrypts it to share it with another peer.

Owl generates an internal checksum when reading chunks from external sources, passes the checksum with the chunk data, and validates chunks with the checksum before returning them to clients. Many Owl clients generate end-to-end hashes when writing to external storage. These clients can optionally provide Owl an integrity checker class containing these hashes and the hash calculation function to validate that the data being read is the same as what they originally wrote. Owl calls the integrity checker as data is being written to a disk or a memory buffer. It fails the download if the integrity checker reports that the calculated hash does not match the write hash.

2.10 Virtual superpeers

One of our original design principles for Owl was that peers should not fetch content that their clients do not read. This led to high network efficiency and made it easier to convince users to adopt Owl since their clients would not be doing work for other services.

However, one recent bucket demonstrated a drawback with this approach. For this bucket, reducing load on external storage is crucial; if data is read too fast, the external storage system throttles readers and performance degrades rapidly. Periodically, a new search index is generated and distributed, which each client then reads at a random time over the next few hours. The first client reads the index directly from external storage, but its memory cache fits only a few chunks. The next client reads those chunks from the first client, and it reads the remainder of the index from external storage. As more clients download the index, their collective caches are eventually sufficient to hold all the data (especially since we use the least-rare caching policy to maximize hit rate for the bucket). However the clients together read many extra chunks from external storage until the index is fully cached, and this causes the external storage system to throttle readers.

To solve this problem, we added a new Owl abstraction called a *virtual superpeer*. If a bucket is configured with a virtual superpeer, the tracker divides each peer's cache into a normal portion and a portion reserved for the virtual superpeer. The tracker aggregates the virtual superpeer portions and manages the collection in the same way that it would manage a superpeer dedicated solely to the bucket. When the first client reads the index, the bucket selection policy routes the ephemeral distribution tree for each chunk through the virtual superpeer. The tracker uses the per-bucket policy to select one peer to fetch the chunk from external storage and cache it; the tracker also selects chunks to evict from the virtual superpeer portion of that peer's cache, if necessary. The requesting peer streams each chunk from the peer that fetched it from the external source. After the index is loaded by one peer, the next peer to fetch the index finds all chunks in the virtual superpeer cache. Thus, Owl makes no additional reads to external storage, and it achieves a high cache hit rate.

The benefit of virtual superpeers over non-virtual (physical) superpeers is that virtual superpeers use spare memory capacity on peers rather than dedicated machines. The bucket described in this section would require approximately 640 physical superpeers to achieve the same cache hit rate as Owl achieves with virtual superpeers. Another bucket that we are currently onboarding would require approximately 10,000 physical superpeers; we are avoiding this cost by leveraging spare peer memory via virtual superpeers.

Virtual superpeers are a tracker-only concept; peers are unaware of the abstraction because they simply follow tracker instructions for where to fetch data and which chunks to cache. Further, the abstraction is implemented almost entirely via Owl selection and caching policies (we added a few hundred lines of tracker code to implement cache partitioning and eliminate double-buffering). Overall, virtual superpeers demonstrate the flexibility of Owl policies: we were able to implement a substantial change not envisioned in the original Owl design primarily by writing new policies.

2.11 Ephemeral data sources

Owl was originally designed to download content from external storage. However, several clients wanted to use Owl to distribute content produced by instances of their service directly to other instances, bypassing storage entirely. For AI models and search indexes that have diminishing value over time, durable storage provides little benefit. Yet, the resources used to read and write large data objects to distributed storage can be significant. We modified Owl to support these use cases by adding *ephemeral data sources*.

An ephemeral data source is simply a peer that promises to supply specific content when requested. The client calls `provide_file` in Table 1 to specify a file containing content for a given unique identifier. In turn, the peer tells the tracker that this content is now in its cache. When other peers requests chunks from this content, the tracker builds ephemeral distribution trees rooted at the providing peer to distribute the chunks. The tracker also advertises and provided content to other sharded trackers, which makes the content available via delegation.

Owl guarantees that the data will be provided only as long as an ephemeral data source provides the data. It caches ephemeral content on peers and superpeers as normal, and it falls back to the peer(s) providing the data as a last resort. Ephemeral data sources must re-register with a new tracker if their connection with the current tracker fails; they send heartbeats every second to their trackers to proactively detect failures and re-register quickly. A client may stop providing content by calling `evict_file`.

2.12 Fault tolerance

Tracker sharding allows Owl to tolerate tracker faults. When a peer detects that its tracker has failed, it re-registers to use a new tracker. Trackers have only soft state, and a new tracker learns a peer's existing cache state as part of registration. We regularly test failover by continuously deploying tracker code each workday, during which trackers are sequentially killed and restarted. We occasionally experiment by killing and restarting all sharded trackers simultaneously to ensure that performance does not drop below SLA bounds when all trackers restart. This has proved enlightening; e.g., we added peer re-registration when we noticed that SLA bounds had been violated during one such trial.

The RPC routing layer at Meta (not part of Owl) load balances requests among sharded trackers at the granularity of each new chunk download. However, peer associations with trackers are typically long-lived, as rebalancing does not need to be done often in steady state. In contrast, Owl load balances superpeers among trackers itself because the number of superpeers per tracker is small and we want to maintain a tighter balance than the RPC router layer provides.

Trackers detect peer failures when a peer reports that it cannot reach a peer from which it is trying to get data. Peers are marked down (and not used to serve further requests) after a configurable number of consecutive failures. Peers are marked up again when they re-register with a tracker. The Owl library explicitly deregisters on shutdown, but many peers don't shut down cleanly, in which case there is no deregistration.

Generally, we do not allow peers to fail over and use trackers outside their region. Experiences with other systems left us concerned about cascading failures in which a failure in one region causes out-of-region requests to overload services in other regions. We use a separate set of global trackers for buckets that require peers to contact out-of-region trackers.

2.13 Emulation and customization

Over time, Owl has become more customizable as we have added new policies and enabled different behavior via configuration within each policy. This flexibility often makes it difficult to determine the best set of policies for each bucket.

The Owl team writes all policies and helps Owl users pick the best policy for their needs. If the team identifies a specific need not covered by an existing policy, we write a new policy; the development of the virtual superpeer policy, described in Section 2.10, is a good example of this process.

Choosing the best policy is difficult. Many engineers who wish to use Owl do not understand their service traffic patterns well. In some cases, it is not clear whether their workload would benefit from peer-to-peer distribution. In other cases, it is difficult to choose the best set of policies or explain specific configuration tradeoffs; e.g., how much additional cache hit rate can the bucket expect for each additional gigabyte of peer memory used? For existing users of Owl, traffic patterns and distribution goals change over time (e.g., a service can spare less memory or require better cache hit rate to reduce external storage load). Thus, initial policy choices often need to be tuned to keep pace with client changes. As the number of buckets using Owl grew, it became infeasible for the Owl team to manually choose and tune policies for each unique workload.

Owl uses offline, trace-driven emulation to guide policy choices. On a per-bucket basis, Owl can be configured to log basic information about each client request to a database; e.g., the timestamp of the request, the object, and data range read. When we onboard a new bucket that reads data from an external source, we first use an *evaluation mode* policy that always instructs the peer to fetch from the external source and not cache data. The peer thus performs the same actions it would perform without Owl except that each request is routed through a tracker for logging. For existing buckets, logging is always enabled.

An Owl emulator runs our actual tracker service with mock peers and superpeers that generate traffic and service requests. The emulator is event-driven and uses a virtual clock to determine when events occur. Mock peers register, deregister, and generate requests at the times recorded in the production traces. The emulator adds configurable network and storage delays, and can simulate different error profiles. Because we run the actual tracker code, we can emulate any Owl policy or configuration. The emulator reports key statistics such as overall cache hit rate, load on external storage, and tracker CPU usage.

To find the best policy, we compare statistics from multiple emulation runs with the same trace and different policy/configuration settings. As the setting space is quite large, we use random-restart hill climbing [10] to search for the best choice for each bucket. Currently, we run the emulator weekly on existing Owl buckets, and we use the emulator to evaluate all new buckets during onboarding.

3 Evaluation

Our evaluation answers the following questions:

1. How well does Owl provide hot content distribution in production?

2. How does Owl compare to other centralized and decentralized distribution solutions?

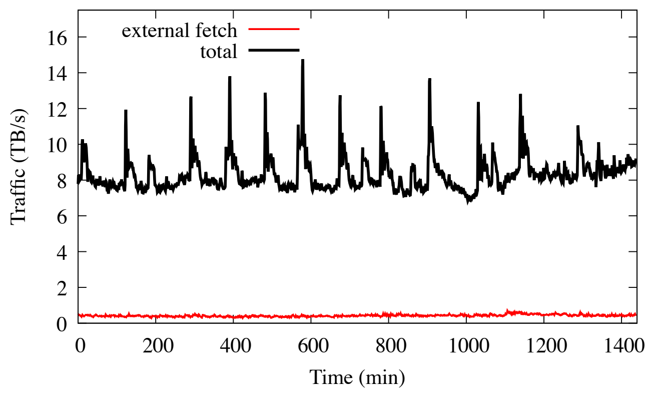

Figure 4: **Owl traffic over a 24 hour period.** The top line shows total bytes downloaded by all clients averaged every minute. The bottom line shows the total bytes read from external storage. The difference between the two lines is the reduction in external storage load due to Owl peer-to-peer caching and distribution.

3. How much benefit does Owl realize from delegation?
4. What are the benefits of flexible distribution policies?
5. How well does Owl scale and how many peer resources does it require?

3.1 Reducing load on external storage

Figure 4 shows a recent (and typical) 24 hours of Owl production traffic, aggregated by minute. The top line is the amount of data read by clients; this is the load they would impose on external storage without Owl. The bottom line shows the load on external storage with Owl. During the 24 hours, Owl clients read 717 PB of data, yet only 36.5 PB was read from external storage, for a cache hit rate of 94.9%.

The cumulative read rate across all Owl clients varies from a minimum of 6.84 TB/s to a maximum of 14.75 TB/s. Figure 4 shows that peer-to-peer distribution and caching hides the client load spikes almost entirely from external storage; in fact, the load on external storage never exceeds 0.72 TB/s.

A CDN or hierarchical caching could also reduce the load on external storage, as in a recently reported study of CacheLib [3]. In that study, each caching node could sustain a maximum data rate of approximately 640 MB/s. Thus, even assuming perfect load distribution, it would require over 23,000 caching nodes to handle Owl's peak client request rate for the reported period, which is more that 200 times the number of current Owl trackers (112).

Figure 5 shows the scalability benefit of Owl's decentralized data plane by comparing the relative growth in production traffic and servers (trackers and superpeers) in 2021. While Owl's peak traffic is almost 200 times greater than traffic at

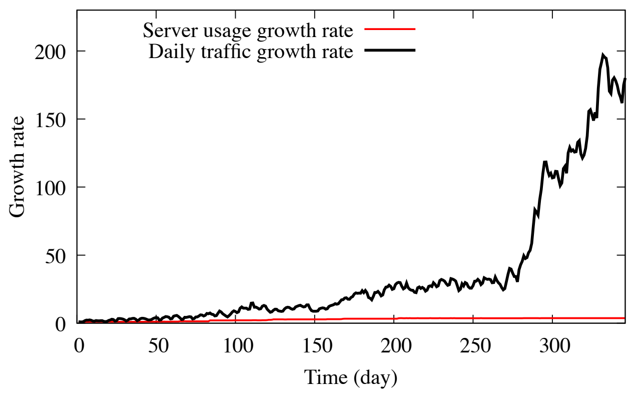

Figure 5: **2021 growth in traffic and server usage.** The top line shows Owl's daily 2021 traffic load relative to the load at the beginning of the year, and the bottom line shows the number of servers (trackers and superpeers) used in production relative to the number used at the beginning of the year.

the beginning of the year, server usage has grown by less than a factor of 4.

When Owl replaced hierarchical caching solutions at Meta, we also saw latency speedups from 50% to 100% for several large buckets due to better network locality and elimination of throttling errors.

3.2 Benefits of delegation

Figure 6 shows the number of successful and failed delegation requests for all Owl trackers over the same 24 hour period. 97.5% of delegation requests are successful; they return a peer that provides the requested content. The primary reason why delegation requests fail is because a sharded tracker's list of cached objects is stale. The low rate of delegation failure indicates that a 1 second update interval is sufficient for the majority of our workloads. We verified this by reducing the delegation interval to 250 ms in one region for 24 hours. Owl's largest bucket saw only a 1% improvement in cache miss rate, and overall cache miss rate did not improve within experimental error.

Delegation provides 10.1% of the total data read by Owl in the 24 hour period. In other words, without delegation, the Owl cache hit rate would decrease from 94.9% to 85.4% (increasing the miss rate by nearly a factor of 3). Delegation is thus an essential factor in providing good download efficiency with sharded trackers.

3.3 Comparison with prior systems

We next compare Owl with the two peer-to-peer distribution systems it replaced at Meta. The first such system was a location-aware implementation of BitTorrent. We configured roughly half the hosts in one region to download binaries

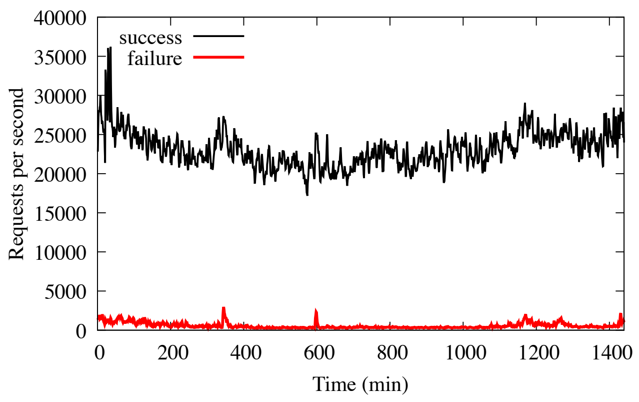

Figure 6: **Delegation success rate over a 24 hour period.** The top line shows the number of successful delegation requests, and the bottom line shows the number of unsuccessful delegation requests.

for provisioning via Owl; the remaining half used BitTorrent. Both systems had identically-sized disk caches. During a 24 hour experiment, these hosts downloaded 17.6 million binaries with a median size of approximately 300 MB. Table 2 compares results for the two systems. Owl is significantly faster than BitTorrent, almost doubling the median per-client download throughput and more than quintupling the p95 throughput. Because the client writes downloaded binaries to local storage, the maximum throughput of Owl was often capped by the available write bandwidth of local media on each host; in contrast, BitTorrent rarely reached the storage bandwidth limit. Additionally, Owl reduces the load on external storage by 42% due to its higher cache hit rate (99.21% for Owl and 98.64% for BitTorrent). Both Owl and BitTorrent provided 4 9's of availability. BitTorrent had slightly higher availability due to allowing more retries with additional backoffs; we later adjusted Owl's retry policies for this bucket to more closely match BitTorrent's policy.

The second prior download system at Meta, StaticTree, used a relatively-static distribution tree constructed via consistent hashing. Each chunk has one primary cacher that fetches the chunk from storage and caches it. The primary cacher is determined by hashing the unique chunk id and selecting a host from a membership list stored in Zookeeper [9]. Each tree level corresponds to a location type (e.g., region, data center, rack, etc.) with the node at each level and location responsible for a chunk again selected via consistent hashing. Secondary nodes at each level provide fault tolerance.

Table 3 compares important download metrics for Owl and StaticTree. Experiments ran for 1–7 days and consisted of millions of production requests to both systems. Both systems use identically-sized memory caches. Owl provides 4 9's of availability in 3 of the 5 experiments and 3 9's in the remaining experiment. StaticTree provides substantially lower availability because of the time needed to detect and route around failed nodes in the tree, as well as the need to remove failed nodes from the membership list in Zookeeper. In contrast, with Owl, ephemeral distribution trees let trackers avoid using a peer immediately for new chunk downloads as soon as that peer is suspected of being unhealthy or slow.

Compared to StaticTree, Owl improves p50 download latency by an average of 55% and p99 latency by 32% across the five experiments. While Owl's latency improvement comes partially from better failure handling, the improved latency also results from the tracker dynamically picking the best data source for a chunk on each *getSource* request. Trackers improve latency by considering load on peers and network locality based on detailed peer and chunk state.

Improving per-chunk download latency often translates into even greater improvements for application-level metrics. Table 4 compares the average time to load six different types of AI models in production via Owl and StaticTree. Owl speeds up model loading time from 1.44x to 3.48x, for an average speedup of 2.92.

Cache hit rates are roughly equivalent for the two systems (StaticTree provides better cache hit rate in 3 out of 5 experiments, but Owl's cache hit rate improvement in Bucket D is by far the most substantial). We also examined network locality for peer-to-peer data transfers between the two systems; we found locality to be roughly the same as both systems optimize for this metric.

3.4 Optimization results

Owl currently has 106 buckets that use 55 distinct policies and configurations. We use the Owl emulator to regularly search for potential policy improvements. Table 5 shows the optimizations that we found in the previous month. We report savings in either peak or total storage usage over 24 hours that we achieved by modifying bucket policies in production. All of these buckets were seeing throttling from external storage at the time, so reducing storage usage was an important goal.

For the first two buckets, emulation lets us inform bucket owners how much improvement in cache hit rate they could expect from allocating more peer memory to Owl caching. Because these clients had memory to spare, we were able to achieve a substantial reduction in peak load. For the remaining three buckets, we achieved better cache hit rate without the need for any additional peer resources simply by changing the policies used by the tracker to manage each bucket.

3.5 Overheads

We measured Owl overhead on peers by profiling one thousand hosts during production usage. Owl's CPU overhead is only 0.05% on 26-core Intel Cooper Lake processors. Owl allocates memory for data caches and for network buffers; both uses of memory are configurable and controlled by the per-bucket policy depending on the client's tradeoff between

	Availability	Cache hit rate	Per-host throughput (MB/s)		Latency (s)	
			p50	p95	p50	p99
Owl	99.994%	99.21%	130.1	20.17	2	132
BitTorrent	99.996%	98.64%	66.9	3.89	4	255

Table 2: **Comparing download metrics for Owl and BitTorrent.** Both systems are used side-by-side to download binaries in one region for 24 hours. We compare the percentage of successful downloads (availability), the reduction in load on external storage (cache hit rate), the median and 95th percentile throughput (download rate), and the median and 99th percentile download latency for five different buckets.

Bucket	Experiment duration	Downloaded bytes	System	Availability	Cache hit rate	Latency (s)	
						Average	p99
A	7 days	14 PB	Owl	99.99%	85%	46.7	47.9
			StaticTree	99.60%	86%	72.2	78.2
B	1 day	30 PB	Owl	99.99%	99.34%	48.8	107.5
			StaticTree	99.91%	99.50%	51.48	122.8
C	1 day	1 PB	Owl	99.99%	69.47%	114.7	507.9
			StaticTree	99.99%	72.52%	180.5	630.8
D	7 days	22 PB	Owl	99.91%	92.70%	44.8	119.0
			StaticTree	99.83%	81.87%	99.8	128.4
E	7 days	50 PB	Owl	99.96%	99.63%	8.5	69.1
			StaticTree	99.95%	99.47%	13.3	112.1

Table 3: **Comparing download metrics for Owl and StaticTree.** Both systems are used side-by-side in production. We compare the percentage of successful downloads (availability), the reduction in load on external storage (cache hit rate), and the median and 99th percentile download latency for four different buckets.

Model	Loading Latency (sec.)		Speedup
	Owl	StaticTree	
A	31	97	3.13
B	138	199	1.44
C	78	264	3.38
D	75	261	3.48
E	82	282	3.44
F	137	465	3.39

Table 4: **Latency improvement in AI model loading** Each row compares the average loading time using StaticTree with the average loading time using Owl for a different bucket.

performance and resource usage. Outside of these two uses, Owl uses less than 0.01% of RSS (resident set size) memory on hosts with 64 GB memory. For comparison, StaticTree uses 0.15% CPU and 0.03% memory for roughly the same workload, which is 3x the resources used by Owl.

To verify scalability, we ran a load test with Owl trackers running on hosts with 64 GB memory, a 26-core Intel Cooper Lake processor, and a 25 Gb/s NIC. Our load tests showed that each such Owl tracker can support 2.4 TB/s client traffic when the Owl cache hit rate is 99%, or 1.5 TB/s client traffic when the Owl cache hit rate is 70%. The trackers are CPU-bound at these traffic levels. As an additional confirmation of being CPU-bound, over 30 days of operation, tracker memory (RSS) stayed at 11% or less on 64 GB hosts, while CPU usage spiked up to a maximum of 37%. In practice, Owl uses many more trackers than these numbers would indicate to provide redundancy for failures, regional failure isolation, and performance isolation among critical buckets.

4 Related work

BitTorrent is the most widely-recognized solution for peer-to-peer data distribution. Classic BitTorrent [7] is highly-decentralized; the trackers simply help peers find each other. Trackers originally returned a random list of peers containing desired data, but later BitTorrent implementations introduced refinements. For instance, the BitTorrent version at Meta sorts peers by location before returning the list. Many recent BitTorrent versions replace trackers with a decentralized distributed hash table [14] for so-called trackerless torrent.

Recently, peer-to-peer distribution has been used to provision containers and virtual machines in public and private clouds. Some implementations have used BitTorrent directly [5]. Uber's Kraken [1] uses a BitTorrent-like architecture to provision containers. Kraken uses trackers that returned an ordered list of candidate peers for downloading data, and it uses dedicated seeders to read from external storage. Alibaba's Dragonfly [2] also provides peer-to-peer container provisioning. Dragonfly's SuperNodes combine tracker and

Bucket	Metric	Before	After	Optimization
A	Peak storage usage	124 GB/s	54 GB/s	Increase peer cache size from 0.2 GB to 4 GB
B	Peak storage usage	63 GB/s	27 GB/s	Increase peer cache size from 0.2 GB to 4 GB
C	Peak storage usage	95 GB/s	18 GB/s	Change selection policy from location-aware to hot-cold
D	Daily storage usage	1.7 PB	1.3 PB	Change eviction policy from LRU to least-rare
E	Daily storage usage	11.7 PB	10.7 PB	Change eviction policy from LRU to least-rare

Table 5: **Savings from continuous offline analysis.** 5 production buckets were optimized in a 1 month period baed on offline analysis results. The table shows the key metric being optimized and the value of the metric before and after optimization.

seeder functionality.

FaaSNet [17], also from Alibaba, takes an even more decentralized approach for distributing serverless containers, foregoing the use of all centralized nodes and instead utilizing a tree-based network overlay. Dadi [13] and VMThunder [19] also use a tree overlay to distribute container/VM images among peers, with cache misses serviced by nodes higher in the tree. These approaches are similar to Meta's StaticTree, except that StaticTree constructs locality-aware trees that minimize the network distance between sibling nodes.

Classically, many tree- and mesh-based networks have been proposed for high-bandwidth data distribution to many hosts [4,6,11,12]. While Owl uses a forest of distribution trees, these trees are per-data-chunk and ephemeral, with edges persisting only for the time needed for a peer to download a single chunk of data. The trees in prior works are longer-lasting and used for more than just a single data chunk.

In contrast to all of these prior distribution systems, Owl's control plane is significantly more centralized. Owl trackers make explicit decisions on behalf of peers about what data to cache and evict, where to download each chunk from, and how to retry failed downloads. Owl trackers consequently have a much more complete view of download and peer state, which allows for making more optimal, global decisions. Centralization also improves ease-of-management. The classic argument against centralizing the distribution control plane has been a projected scalability bottleneck; Owl refutes this argument by demonstrating that careful design can scale even a highly-centralized control plane to support millions of clients and hundreds of petabytes of data distributed per day. Additionally, Owl demonstrates more flexibility than prior systems through its customized policies; container provisioning is currently just a small portion of Owl's total workload.

The control plane and data plane taxonomy used in this paper comes from software defined networking (SDN). Recently, Google's Orion [8] demonstrated the benefits of centralizing the SDN control plane. Distribution and SDNs both provide routing and store-and-forward-style caching. Yet, Owl tracks each individual chunk of data at a level of detail that is infeasible for network packets. This is possible because Owl operates on much larger units of data.

5 Conclusion

Owl distributes over 800 PB of hot content per day to millions of peers at Meta. Owl combines a decentralized peer-to-peer data plane with a highly-centralized control plane in which trackers make detailed decisions for peers such as for where to fetch each chunk of data, how to retry failed fetches, and which chunks to cache in peer memory and storage. Owl is highly-customizable through tracker policies that allow a unique configuration for each type of client.

6 Acknowledgements

We thank our shepherd, Doug Terry, and the anonymous reviewers for providing valuable feedback that allowed us to improve this paper. We also thank CQ Tang for reading early iterations of the paper and helping us focus our discussion.

References

[1] Introducing Kraken, an open source peer-to-peer docker registry. https://eng.uber.com/introducing-kraken/.

[2] What is Dragonfly? https://d7y.io/en-us/docs/overview/what_is_dragonfly.html.

[3] Berg, B., Berger, D. S., McAllister, S., Grosof, I., Gunasekar, S., Lu, J., Uhlar, M., Carrid, J., Beckmann, N., Harchol-Balter, M., and Ganger, G. R. The CacheLib caching engine: Design and experiences at scale. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (November 2020).

[4] Castro, M., Druschel, P., Kermarrec, A.-M., Nandi, A., Rowstron, A., and Singh, A. Splitstream: High-bandwidth multicast in cooperative environments. In *Proceedings of the 19th Symposium on Operating Systems Principles (SOSP)* (October 2003).

[5] Chen, Z., Zhao, Y., Miao, X., Chen, Y., and Wang, Q. Rapid provisioning of cloud infrastructure leveraging peer-to-peer networks. In *Proceedings of the*

9th IEEE Conference on Distributed Computing Workshops (2009).

[6] CHU, Y.-H., RAO, S. G., SESHAN, S., AND ZHANG, H. Enabling conferencing applications on the internetusing an overlay multicast architecture. In *Proceedings of ACM SIGCOMM* (August 2001).

[7] COHEN, B. Incentives build robustness in BitTorrent. In *Proceedings of the Workshop on Economics of Peer-to-Peer Systems* (2003).

[8] FERGUSON, A. D., GRIBBLE, S., HONG, C.-Y., KILLIAN, C., MOHSIN, W., MUEHE, H., ONG, J., POUTIEVSKI, L., SINGH, A., VICISANO, L., ALIMI, R., CHEN, S. S., CONLEY, M., MANDAL, S., NAGARAJ, K., BOLLINENI, K. N., SABAA, A., ZHANG, S., ZHU, M., AND VAHDAT, A. Orion: Google's software-defined networking control plane. In *Proceesings of the 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (April 2021).

[9] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. ZooKeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the USENIX Annual Technical Conference* (2010).

[10] JACOBSON, S. H. Analyzing the performance of local search algorithms using generalized hill climbing algorithms. *Essays and Surveys in Metaheuristics 14* (2002), 441–467.

[11] JANNOTTI, J., GIFFORD, D. K., JOHNSON, K. L., KAASHOEK, M. F., AND O'TOOLE, J. W. Overcast: Reliable multicasting with an overlay network. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (October 2000).

[12] KOSTIC, D., RODRIGUEZ, A., ALBRECHT, J., AND VAHDAT, A. Bullet: High bandwidth data dissemination using an overlay mesh. In *Proceedings of the 19th Symposium on Operating Systems Principles (SOSP)* (October 2003).

[13] LI, H., YUAN, Y., DU, R., MA, K., LIU, L., AND HSU, W. DADI: Block-level image service for agile and elastic application deployment. In *Proceedings of the 2020 USENIX Annual Technical Conference* (July 2020).

[14] MAYMOUNKOV, P., AND MAZIERES, D. Kademlia: A peer-to-peer information system based on the xor metric. In *Proceedings of the International Workshop on Peer-to-Peer Systems* (2002), pp. 53–65.

[15] TANG, C., KOOBURAT, T., VENKATACHALAM, P., CHANDER, A., WEN, Z., NARAYANAN, A., DOWELL, P., AND KARL, R. Holistic Configuration Management at Facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)* (Monterey, CA, Oct. 2015).

[16] TANG, C., YU, K., VEERARAGHAVAN, K., KALDOR, J., MICHELSON, S., KOOBURAT, T., ANBUDURAI, A., CLARK, M., GOGIA, K., CHENG, L., CHRISTENSEN, B., GARTRELL, A., KHUTORNENKO, M., KULKARNI, S., PAWLOWSKI, M., PELKONEN, T., RODRIGUES, A., TIBREWAL, R., PAWLOWSKI, M., PELKONEN, T., RODRIGUES, A., TIBREWAL, R., VENKATESAN, V., AND ZHANG, P. Twine: A Unified Cluster Management System for Shared Infrastructure. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI) 2020* (November 2020).

[17] WANG, A., CHANG, S., TIAN, H., WANG, H., YANG, H., LI, H., DU, R., AND CHENG, Y. FaasNet: Scalable and fast provisioning of custom serverless container runtimes at Alibaba cloud function compute. In *Proceedings of the 2021 USENIX Annual Technical Conference* (July 2021).

[18] YANG, J., YUE, Y., AND RAHMI, K. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (November 2020).

[19] ZHANG, Z. Z., LI, Z., WU, K., LI, D., LI, H., PENG, Y., AND LU, X. VMThunder: Fast provisioning of large-scale virtual machine clusters. *IEEE Transactions on Parallel and Distributed Systems 25*, 12 (2014), 3328––3338.