TaskMaster: A Scalable, Reliable Queuing Infrastructure for Building Distributed Systems

Josh Hyman
joshh@google.com
Google Inc.

Dustin Boswell
dboswell@google.com
Google Inc.

Abstract

TaskMaster is a system for managing priority-ordered queues that is designed to scale to 1 billion tasks across 100 thousand queues per node. A reliable queuing system, such as TaskMaster, provides a mechanism for distributing units of inherently serial work (tasks) to workers. Priorities are lexicographically ordered strings that give users more power than FIFO or fixed-range integer priorities when defining queue order. TaskMaster provides the ability to atomically transition a task between queues, facilitating the decomposition of processing into a sequence of tasks. Individual tasks are exclusively leased to workers for the duration of processing to minimize system-wide work duplication. TaskMaster helps system designers detect bottlenecks and direct system optimization by providing queue statistics such as enqueue and dequeue rates, as well as queue lengths. TaskMaster allows application designers to focus on the individual stages of data processing instead of task distribution.

1 Introduction

There exists a large class of problems which consist of many independent and inherently serial units of work. The transactional semantics of these problems are such that they can typically be broken down into a sequence of idempotent sub-problems [9]. For example, extracting and indexing content from a web page first requires fetching the document itself, then the document must be processed to gather the meaningful data, and finally, these data must be added to an index. Each of these steps are idempotent and take significant time to complete either because of computational expense or network delay.

Such problems can be parallelized by exploiting the large number of tasks to be completed and the implicit pipeline structure of computation. TaskMaster simplifies the distribution of individual sub-problems by joining stages of processing together using atomic operations on reliable queues. Typical applications built using TaskMaster consist of a set of *worker pools*, which are sets of processes that are prepared to perform a predefined operation on tasks that are leased from TaskMaster. Applications use a pull-based model where workers pull tasks from TaskMaster eliminating the need for TaskMaster to keep track of workers in the system. This greatly simplifies the implementation of TaskMaster itself. It also allows worker pools to be dynamically resized in response to load or other factors without notifying TaskMaster.

TaskMaster has been in continuous use for over a year, facilitating the construction of both continuous latency-sensitive and batch throughput-oriented applications. These systems range from having a single TaskMaster node and a few workers, to having a multi-node TaskMaster cluster with hundreds of workers. These disparate distributed applications show the flexibility of TaskMaster and the diverse applicability of the reliable queuing model.

Section 2 provides a more detailed description of TaskMaster's data model. In Section 3 we explain the various design decisions made to enact this model, and Section 4 provides an overview of the client API used by workers. Then, Section 5 describes a basic implementation of TaskMaster and a set of important refinements to improve performance. Next, Section 6 discusses the performance of the system and the effects of the various optimizations. Section 7 illustrates how TaskMaster is used in various applications at Google. Finally, Section 8 describes related work and Section 9 draws conclusions.

2 Data Model

TaskMaster manages a set of named priority-queues that contain tasks. A task is a *(queue, priority, data)* triple of user-provided binary strings representing an opaque unit of idempotent work from TaskMaster's perspective.

The queue name is a pair of *(consistency group, name)*; if no consistency group is provided, the default consistency group is used. Operations within a consistency group are atomic (Section 3.4). Priorities are lexicographically ordered and are unique within the named queue. The data field is not interpreted by TaskMaster and has no a priori size limit; but for performance reasons is typically between 10 bytes and 10KB. Queues themselves are ephemeral in that they do not exist unless they contain tasks.

Beyond supporting enqueue and dequeue operations of a normal queue, TaskMaster supports exclusive, arbitrary-length leasing of tasks to clients. Exclusive leases ensure workers process disjoint sets of tasks with minimal coordination. Using the lease operation, clients can discover and lease the top-k highest priority tasks from a given queue. Upon lease expiration, tasks are free to be re-leased to other clients.

3 Design

TaskMaster is designed to work in single or multiple node configurations. Each node can manage *O(1B)* tasks across *O(100K)* named queues. Workers are provided with an interface through which they can discover the names of queues as they are not expected to have a priori knowledge of these names. Atomic update operations are provided to allow workers to reliably move a unit of work to the next stage of processing. TaskMaster attempts to fairly divide resources among queues and ensure liveness to workers performing operations on those queues.

To aid administration, TaskMaster provides efficient queue deletion and queue statistics such as size, enqueue rate, dequeue rate, and so on. In this section we discuss how each of these features is designed and the reasoning behind those decisions.

3.1 Assumptions

The TaskMaster cluster and associated application specific workers are assumed to run in a datacenter environment with a relatively high bandwidth and low latency connectivity. All nodes are expected to be relatively powerful, heterogeneous commodity PCs as described in [4]. We anticipate node failures of both workers and TaskMaster nodes to be caused by power failure, hardware failure, or those jobs being forcibly rescheduled by the cluster management application. Thus, node failures are typically fail-stop. Additionally, we expect occasional network partitions between TaskMaster and its workers, and between TaskMaster and its backing data store.

The data store backing TaskMaster is only expected to provide durability and atomicity of single writes; atomicity, consistency, and isolation across multiple writes are provided by TaskMaster itself. In particular, the data store is free to reorder the application of mutations as long as writes are durable once it responds. The data store need not support multi-row transactions.

3.2 Uses of Priority-IDs

Named Priority-IDs (PIDs) because they serve as both a priority within a queue and a unique task identifier, PIDs provide users more flexibility than a simple first-in, first-out (FIFO) queue. We chose to allow arbitrary PIDs because of the large variety of use-cases it enables. Most importantly, PIDs allows an application to give particularly important tasks high priority.

A simple FIFO queue can be approximated with PIDs using the local time at the client plus a locally accessible unique value (like CPU ID or MAC address). However, for efficiency, TaskMaster also provides explicit support for FIFO queues when the user does not provide a PID. In these cases, TaskMaster keeps track of the last PID in the FIFO queue and assigns a lexicographically larger (and thus unique) PID to such incoming tasks.

Additional control is useful if applications want to produce an ordering of tasks different from the order in which they were enqueued. For example, when enqueuing order might induce contention on shared resources (like a set of files), PIDs can be a coarse local timestamp plus a hash of the data as the suffix. This produces jitter in the task order while approximating FIFO. Inversely, PIDs are also useful if tasks are enqueued in a random order but some order should be imposed. For example, when reading multiple sections of a file, it would be more efficient to read them in order even if they were requested in random order. Here, the PID could be the file offset of the various chunks.

Another use of PIDs is to limit work duplication when there is a priori knowledge of uniqueness of tasks. For example, if a task was requesting to fetch the HTML for “http://www.google.com”, it could be given the PID “http://www.google.com”. While that task was still queued waiting for processing, future enqueues of the same task would collapse into a single task.

3.3 Task Leasing

The primary purpose of leasing tasks is to limit work duplication in the system. TaskMaster provides exclusive, arbitrary-length task leases to workers instead of locks to make applications more resilient to worker failure. Upon worker failure, the leases it held will simply expire and a new worker can acquire a new lease on those tasks. In some failure modes, this may result in a task being executed more than once, perhaps in parallel, which is allowed because tasks are expected to be idempotent.

For example, suppose a worker leases a task and then disappears because of a network partition. The task may then be expired and given to another worker while the first worker continues to execute the task. Alternatively, a previous worker may have completed the work and failed before notifying TaskMaster.

When requesting a lease, a worker will receive at most k tasks but we do not guarantee those tasks will be the exact k highest priority tasks. Relaxing this constraint allows for increased concurrency (Section 5.4.2) and lazy lease expiration (Section 5.6). This leniency does not add any additional complexity from the user's perspective; receiving an approximate top-k result can be modeled as a reordering of an enqueue and a lease operation acting on the same queue. This is common in practice because workers asynchronously enqueue tasks while others lease them.

In many cases, workers can't effectively estimate the duration of a task's execution. For example, downloading an object of unknown size from the Internet might take arbitrarily long. TaskMaster provides the ability to renew the lease on a given task to avoid the unintentional and exceptionally expensive duplication of this work upon lease expiration. There is no limit on the number of renewals for a given task as this mechanism is only meant to detect worker failure; in this capacity, it acts as a heart-beat signal from the worker. Renews for already expired tasks are ignored and an error is returned to the client.

3.4 Atomic Operations

TaskMaster provides atomic queue operations within a consistency group. A consistency group is a set of queues whose names share a prefix (recall, a queue name is *(consistency group, name)* pairs) and may contain an arbitrary number of queues. All queues in a consistency group are guaranteed to be managed by the same TaskMaster node so that clients do not need to perform a distributed transaction across TaskMaster nodes to achieve atomicity. Thus, consistency groups provide a natural unit of partitioning (Section 5.7) when using TaskMaster in a cluster configuration.

TaskMaster provides several atomic operations within a consistency group, namely lease and update operations. Lease operations allow workers to atomically discover and lease the k highest priority tasks from a given queue. This ensures that workers will work on a disjoint subset of available tasks even when performing these requests in parallel. Update operations contain instructions to enqueue, dequeue, and renew leases on any number of tasks from any number of queues. Atomically dequeuing one task and enqueuing another provides workers with the ability to easily transition a unit of work between stages in a pipeline.

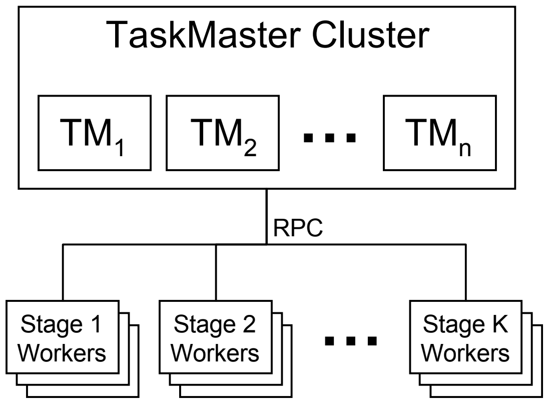


Figure 1: Shows the typical topology of a TaskMaster-based application. Each box is a separate process.

3.5 Queue Introspection

TaskMaster provides several important details about the queues it manages to aid in debugging the distributed applications it enables. First, it provides the enqueue, lease, and dequeue rates to help identify bottlenecks in the distributed system. Second, it provides total queue size and leased set size to highlight queues which are most affected by the previously identified rate mismatch. Third, a sample of leased tasks can be viewed together with the associated worker's IP address to help discern the cause of processing latency. Finally, it provides the average measured lease duration (wall time from lease acquisition until task dequeuing) of tasks from the queues. This represents the average length of time a worker spends processing tasks from a given queue, suggesting stages which could most benefit from optimization.

Most of the data backing the queue statistics is considered soft state because it can easily be regenerated after a TaskMaster failure; thus, it is not stored in the data store. However, queue and leased set size must be accurately accounted for and checkpointed because acquiring this data would require a full scan of the data store.

Since TaskMaster is expected to manage a large number of queues, users can view any of the available statistics on any subset of queues to help focus their debugging. In the case where multiple TaskMaster nodes exist in a cluster, each node is able to query the others to produce an aggregate response across all TaskMaster nodes.

Program 1 Enqueue a task in TaskMaster

```
UpdateRequest request;
Task& t = request.add_enqueued();
t.set_queue("google.com#fetch_stage");
t.set_pid("http://www.google.com");
t.set_data("fetch document");

UpdateReply reply;
tm_stub->UpdateTasks(request, &reply);
assert(reply.success());
```

3.6 Application Design

Applications built on TaskMaster typically consist of multiple pools of workers, as seen in Figure 1, each responsible for a single stage in the computation pipeline. Queues representing a particular stage are named in a predictable way, such as a common suffix. At startup, these individual workers ask TaskMaster for a list of queues of interest based on that application's naming scheme. The worker's inner loop iterates over the queues, leases tasks from that queue, processes them, and then atomically dequeues that task and enqueues the derivative unit of work into the next stage's queue.

To make efficient use of this model, each task should represent a unit of work which takes much more time to complete than lease and subsequently to dequeue. In the case where tasks are logically small in nature, many tasks can be dequeued at once to amortize the dequeuing cost across many tasks. Leases can be acquired from multiple queues at once to reduce the quantity of RPCs to TaskMaster when the number of queues become large.

4 API

Consumers interact with TaskMaster via RPC without a client library other than the RPC stub. This alleviates the need to keep various client libraries, written in different languages, in sync as new updates to the protocol become available.

All updates to TaskMaster are performed via the `UpdateTasks()` method. Program 1 shows C++ code that enqueues a single task into a queue residing in the default consistency group (details omitted for brevity). Tasks are added to the request's set to enqueue via `add_enqueued()`; multiple tasks can easily be enqueued together by invoking `add_enqueued()` multiple times. This task is requesting a worker to fetch a given document. It uses the PID "http://www.google.com" so this enqueue will coalesce with any existing request to fetch the same document. Updates may also contain a set of tasks to dequeue and a set of leases to extend. These sets are accessible by `add_dequeued()` and `add_renew()` respectively.

Program 2 Leasing tasks from a TaskMaster queue

```
LeaseRequest req;
req.set_queue("google.com#fetch_stage");
req.set_lease_secs(300);
req.set_max_tasks(10);

LeaseReply reply;
tm_stub->LeaseTasks(req, &reply);
vector<Task*>& lts = reply.leased_tasks();
// process tasks in 'lts' as required
```

Program 3 Listing known TaskMaster queues

```
ListRequest req;
req.set_max_queues(10);
req.set_queue_re("[a-z.]*#fetch_stage");
req.set_min_tasks(5);

ListReply lr;
tm_stub->ListQueues(req, &lr);
vector<string>& qs = lr.matched_queues();
```

Tasks can be leased from TaskMaster via the `LeaseTasks()` method. Clients are free to specify how many tasks they want to lease as shown in Program 2. Additionally, clients can "filter" the returned tasks server-side by a maximum PID value, since it can be applied to the head of the queue without sacrificing correctness. Other filters which would require arbitrary queue traversal are not available.

Queue names can be discovered by the `ListQueues()` method, since workers are not expected to have a priori knowledge of queue names. Returned queues can be filtered by several properties including queue length and name regular expression as illustrated in Program 3. Unlike filters for leasing tasks, we are able to consider many more queue properties because that metadata resides in memory.

The API also provides simple mechanisms for efficiently deleting a queue, gathering queue introspection data from one or more queues, and batching multiple update or lease operations into a single RPC. Additionally, TaskMaster provides access to queue introspection data visually via HTTP.

5 Implementation

The fundamental challenge of implementation is maintaining performance when the amount of available memory is insufficient to hold all of the tasks. In fact, TaskMaster would only be able to store *O(2M)* tasks in memory given a 1KB average task size and 2GB of memory to use. However, we only anticipate *O(100K)* queues to be managed by a single TaskMaster node. This allows us to keep the various per queue statistics in memory that are needed for queue introspection.

As a result of the memory constraints, we must provide an efficient implementation with incomplete information about tasks in the system. We will first show a correct but inefficient implementation, then we will explain successive optimizations which don't compromise correctness and serve to improve performance. These optimizations are evaluated in Section 6.

5.1 Building Blocks

TaskMaster is built on top of Bigtable [6] and as such, some of the implementation decisions were made with Bigtable's strengths and weaknesses in mind. Specifically, random writes and linear reads (scans) are fast whereas random reads are slow. These attributes are artifacts of how Bigtable stores data in SSTables on GFS [15].

Various other storage systems exhibit properties similar to those of Bigtable, and would benefit from similar optimizations. Relational databases, such MySQL [24], exhibit similar random write and indexed read performance (similar to a scan).

Databases provide multi-row transactions with ACID properties in contrast to single-row transactions provided by Bigtable. However, TaskMaster doesn't require the full power of these multi-row transactions and would likely provide better performance due to its knowledge of application specific consistency requirements.

Other systems like Sinfonia [1], Amazon's Dynamo [11], and Distributed Hash Tables like Chord [27] provide a key-value store interface similar to that of Bigtable. However, these systems don't provide any notion of order across keys (like Bigtable rows) which makes building an inherently ordered data-structure significantly more complex.

5.2 Updating the Data Store

An individual task is stored in a Bigtable row corresponding to its *(queue, PID)* pair. In this way, tasks in a given queue are naturally ordered in the Bigtable rowspace, providing efficient access to a priority-ordered range of the queue via the scan interface. The data in that row is both the user specified binary data and additional TaskMaster metadata describing the task's current state (either available or leased). This could easily be implemented in a relational database by adding an index to the *(queue, PID)* pair for efficient lookup.

To provide correctness, all TaskMaster updates require a read from and a write to the data store. The read discovers the current state of any referenced tasks, determining the validity of the request, and the write updates the state as required. For example, for an enqueue operation, all the referenced *(queue, PID)* pairs must be read to check for existence. If none exist, then they each must be written to record their creation.

In the common case where multiple tasks must be updated, writes are done in parallel. This has the potential to compromise both consistency (upon failure) and isolation between with concurrent requests. We address consistency upon failure with a commit log (Section 5.3) and isolation with lock-based concurrency control (Section 5.5).

To avoid some of the writes, we could choose not to update a task's state when it is leased to a worker. This optimization turns out to be premature as it causes TaskMaster to "forget" the set of leased tasks upon failure. After restart, it would be free to re-lease those tasks to new workers causing an unnecessary duplication of work. This effect is further pronounced for applications which have many tasks leased at once or whose tasks take a long time to complete. Further, to enqueue and dequeue a task requires at least a write and a delete respectively, so removing the intermediate write during a lease has diminished benefit.

Fast queue deletion is done by attaching a generation number to each queue. Upon deletion, the generation number is incremented and all tasks from previous generations of the queue are ignored. These generation numbers are encoded in the task's row name between the queue name and the PID. This has the effect of separating the data backing previous generations of the queue from the currently active queue. A background thread lazily reaps the tasks from the deleted queues as these tasks only use space but don't effect correctness or performance.

5.3 Commit Log

The primary purpose of the commit log is to preserve consistency in the face of fail-stop failures of TaskMaster; each TaskMaster node has its own commit log. The commit log consists of an ordered list of ApplyAtoms, each assigned a sequential ticket produced by TaskMaster. This ticket is written with the ApplyAtom to the data store and the largest used ticket is reconstructed after the commit log is replayed during recovery.

Each ApplyAtom contains a set of enqueues, state changes, and dequeues as well as a set of *(queue name, available delta, leased delta)* triples. These triples serve to persistently store changes to the total queue size and leased set size so they can be efficiently recovered on failure. However, the introduction of these deltas produces a problem: commit log entries are no longer idem-

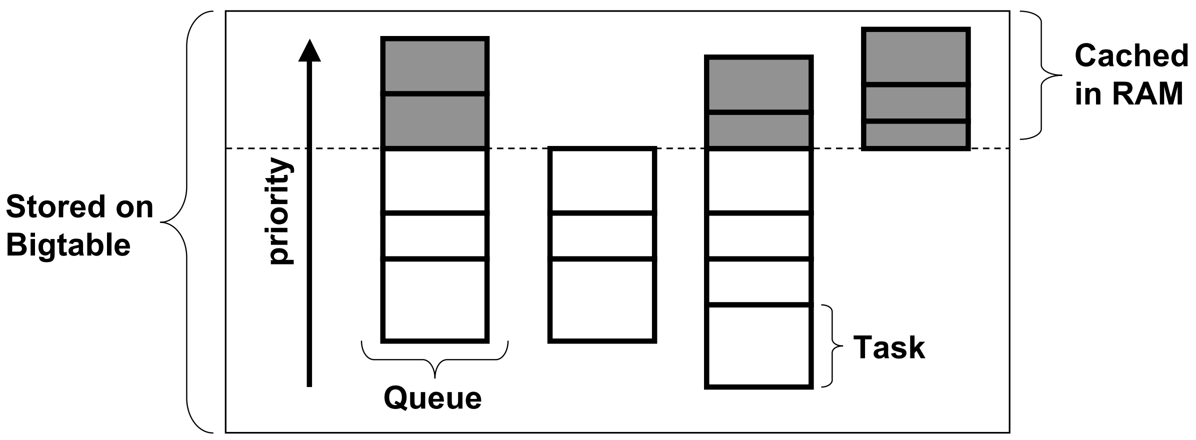


Figure 2: Logical layout of queues and tasks in memory and in Bigtable. Varying task size represents the varying size of PID and data specified by the user.

potent.

To solve this, we built a replay safe counter which expects all update deltas to have an associated monotonically increasing “timestamp”. Thus, any update which occurred in the “past” with respect to the current value’s time is rejected as a replay. The ticket number assigned to an ApplyAtom serves as the timestamp for the counter. These counters then periodically checkpoint their state to the data store and update the minimum ticket which has been successfully applied. The size of checkpoints is minimized by only checkpointing the counters that changed since the previous checkpoint.

ApplyAtoms were originally stored directly in GFS files using the RecordAppend operation [15]. However, this caused intermittent performance hiccups and record reordering due to GFS chunkserver failures. Instead, ApplyAtoms are written to sequential rows in Bigtable identified by their ticket number. We allow ApplyAtoms to be written to the commit log in parallel for efficiency. An ApplyAtom is considered committed once it and all earlier ApplyAtoms have been successfully written.

Recovery is simply a matter of scanning from the minimum applied ticket through the end of the range and reapplying each operation in order. The number of operations which need to be replayed is a function of the number of in-progress operations when the TaskMaster died and frequency with which the minimum ticket is written to the data store. In practice, a few hundred operations are in-progress at once and the minimum ticket is written to the data store every 500ms. This results in fewer than 500 operations to replay on start-up even when TaskMaster is under load. Thus, recovery time is only takes on the order of a few seconds to complete.

5.4 In-Memory Task Cache

The difference in speed between random writes and random reads suggests the use of an in-memory task cache to conserve reads. Figure 2 illustrates the relationship between the task cache and the logical structure of the queues; essentially, the head of each queue is cached. In particular, the entirety of the task (PID and data) is cached. Without caching the data, the cache is not actually saving a read as the client requires that information during a lease request.

Since we are caching a contiguous range of a priority-queue, TaskMaster has full knowledge of all PIDs between the the empty PID (highest priority) and the so-called “first missing PID”. The first missing PID represents the lowest PID in the data store which isn’t in the cache. As tasks get evicted from cache, they place a gap in this knowledge. So, we update this range of PIDs to be bounded below by the evicted PID. This information serves to eliminate reads during enqueues and avoid scanning empty PID ranges upon refill.

The naive implementation of this cache would in fact be multiple caches, one for each queue. In the background, tasks would be optimistically prefetched into those caches. However, ensuring fairness among queues is difficult because the memory allocated to a given queue changes as that queue’s throughput varies and as the total number of queues change. That is, the fairness criteria is not measured in absolute per queue memory allocation, but rather the number of tasks needed to satisfy requests for the period between refills by the background thread. The result is a brittle implementation which suffers from cache starvation for high throughput queues, and excessive memory usage by queues exhibiting bursty throughput.

5.4.1 A unified cache

A simpler implementation would be a single large cache exporting a map-like interface to allow efficient access to the inherent queue structure. Keys into the cache would be a *(queue, PID)* pairs. A least-recently-used (LRU) eviction policy alleviates the need to constantly vary per queue memory allocations. Tasks in queues which idle long enough will eventually be evicted from the cache, correctly allocating memory to active queues. Further, using an on-demand refilling model delays the read from the data store until its absolutely required resulting in fewer wasted reads. The number of prefetched tasks is determined by the historical queue throughput and bounded by a user defined constant to prevent cache thrashing due to many huge refills.

5.4.2 On-demand cache refilling

An on-demand cache refill is triggered when the thread processing a lease request notices that the quantity of cached tasks drops below a threshold. Since the refill is a relatively expensive operation, requiring a scan of a Bigtable row range, care is taken to ensure that only one thread attempts to refill the cache for a given queue to the exclusion of other threads. Multiple threads attempting to refill a given queue can occur when multiple workers issue lease operations on the same queue at once. Once the cache has successfully been refilled, each lease operation is awoken to acquire a lease on a subset of the newly cached tasks in parallel. As a result, the tasks acquired by any given lease request would not be the exact top-k tasks with respect to priority. However, the result of lease requests from all waiting threads would produce the top-k tasks in aggregate.

After a cache refill, we set the first missing PID to be the value of the first PID greater than the set of tasks retrieved from the data store. If TaskMaster gets fewer tasks than expected from the scan of the data store, we assume that we have read the entire queue and a special token is stored to denote this fact. Further, if all tasks were read during the previous refill and have since been completed, future requests to refill the cache can safely be ignored.

5.4.3 Avoid pinning cached tasks

Since tasks which are leased from a given queue are likely to be dequeued relatively soon, its tempting to pin them in the cache. Doing so mitigates the read during dequeue. However, if a worker fails while holding leases on tasks, the TaskMaster will continue to pin those tasks in memory until they expire. Moreover, if many tasks are leased for long durations (or are continually renewed) they could tie up all available memory and starve other queues. As a result, leased tasks are evicted like other tasks. Even still, they are typically available in the cache during dequeuing in practice.

5.5 Increasing Concurrency

To provide correctness, the simplest approach would be to ensure sequential consistency by only processing one operation at a time. This would result in no parallelism and poor performance. Instead, we can use a causal consistency model [17], weaker than sequential consistency, to allow non-causal operations to be processed in parallel [20].

We define update operations to be causal with one another if the set of *(queue, PID)* pairs in each update has a non-empty intersection. That is, there exists some unique task that is updated in both operations. A lease operation is defined to be causal with all other operations on the specified queue only if a queue refill needs to take place. In that case, the operations need to be causal to ensure consistent updates to the in-memory cache. Delete queue requests are defined to be causal with all other operations on the specified queue. Notice that in the common-case, where cache refills and queue deletions are infrequent, only updates which access overlapping *(queue, PID)* pairs are causal; the minimal required causality between events to ensure consistency.

To implement these causal relationships, we use a series of reader-writer mutexes: one for each queue, and one for each *(queue, PID)* pair. All mutexes are ephemeral in that they don't "exist" if no thread has requested a lock (reader or writer) on the given mutex. The implementation of these mutexes prevent writer starvation in the presence of readers [19].

Both queue refill and queue deletion trigger a write lock to be acquired on the affected queue, no other locks need be acquired. Update requests acquire a read lock on the affected queues in order to enforce causality between updates and the other operations. Additionally, updates acquire write locks on all of the affected PIDs to enforce causality between update operations. Lease operations read lock the queue before discovery to enforce causality with other operations. As tasks are discovered their PIDs are write locked to ensure no other concurrent lease will acquire those tasks. We impose a lock hierarchy [12] based on the queue names and PIDs, to avoid deadlock when acquiring all of these locks. Specifically, queues are first locked in lexicographic order then PIDs within a queue are locked in lexicographic order.

The aforementioned locks must be acquired before their respective operations commence and must be held for the duration of the operation (including while data store mutations are outstanding). Since other operations are blocked while a causally dependent operation is in progress, high contention produces high operation latency. It is a tempting optimization to release the acquired locks once in-memory state has been updated. However, we assume that the data store is free to reorder mutations, so we must hold the locks until the data store has confirmed the durability of a write to maintain causal consistency. Releasing the locks early also violates isolation as the latter operation might see the tasks in some intermediate state. The commit log (Section 5.3) is required to ensure the consistency of operations in the presence of TaskMaster failure while an operation is in the process of being committed to durable storage.

5.6 Lazy Lease Expiration

There is clearly no need to expire a worker's lease on a given task if no other workers are interested in leasing tasks from the containing queue. This suggests a lazy expiration policy triggered by lease operations. We specifically perform expiration after refills because leased tasks are not required to be pinned in the cache and thus we may have discovered an expired task during the refill. An advantage of this lazy strategy is that TaskMaster is somewhat forgiving of slight lease violations. For example, if a worker exceeds its lease on a task during processing and tries to dequeue that task, it may still succeed.

As expiration may trigger writes, we decrease the performance impact by limiting the expiration frequency to a user defined maximum, typically 30 seconds. This choice has multiple side-effects. First, it places an effective minimum bound on meaningful lease lengths. Second, and more importantly, high-priority expired tasks may not be re-leased until lazily expired some time-interval later.

This behavior is allowed because lease requests are only expected to acquire leases on some approximation of the top-k tasks. In practice, workers tend to set their task leases pessimistically and the majority of tasks are completed well within their allotted lease period. Thus, lease expiration is rare, happening mostly due to worker failure (the stated goal of lease expirations).

5.7 Scaling

In order to scale TaskMaster beyond a single node, we partition (or shard) the managed queues by consistency group. The hash value of a consistency group's name determines the owning TaskMaster node. This provides a somewhat uniform distribution of consistency groups to nodes. Updates are not allowed to cross consistency group boundaries and the entirety of a consistency group resides on a single TaskMaster node. We avoid the need for a distributed transaction by sharding this way.

To eliminate the need for a client library when TaskMaster is sharded, workers talk directly to many well-known stateless proxies, each exporting the same interface as TaskMaster. Each stateless proxy is aware of the consistency group to cluster node mapping, and is able to route updates to the appropriate cluster node. It is convenient to bundle the proxy with each TaskMaster node, but since they are stateless, they could conceivably be run anywhere.

Cluster-wide throughput scales well since each TaskMaster node in the cluster is functionally independent. Currently, additional nodes are added to a TaskMaster cluster by forcibly restarting all jobs with a new static sharding. In practice, this form of resharding is acceptable because TaskMaster can typically experience brief periods of downtime without any user-visible effect.

Dynamic cluster resizing could be implemented by having all cluster nodes participate in a membership protocol similar to the tablet server membership protocol in Bigtable. Simpler yet would be a three round membership protocol, as implemented in [26], which avoids the need for a central master. We can use consistent hashing [18] to assign groups to nodes, to minimize the number of consistency groups which need to be moved when adding or removing a node.

An application can achieve increased enqueue availability by evenly distributing the addition of work among multiple TaskMaster nodes or even multiple TaskMaster clusters. Removing a single point of failure from the enqueuing path in this fashion is most important for delay-intolerant applications when the enqueuing operation can be triggered by a user action. After the initial task has been enqueued, the rest of processing would proceed as normal. Most applications which can be mapped onto TaskMaster can leverage this form of partitioning to gain availability.

6 Performance

To measure the performance of TaskMaster, we use a single TaskMaster node being accessed by N worker processes with K threads per process. Each worker process confines its operations to a set of queues disjoint from the queues accessed by other worker processes. However, threads in each process produce operations on the same queue and perhaps even the same PID. Each worker in these tests represents a pool of workers enqueuing and/or processing tasks from a single logical application-defined stage. We only test a single node because we are particularly interested in single server performance; the performance of a TaskMaster cluster depends directly on the performance of individual nodes.

The TaskMaster is typically allocated 1GB of memory

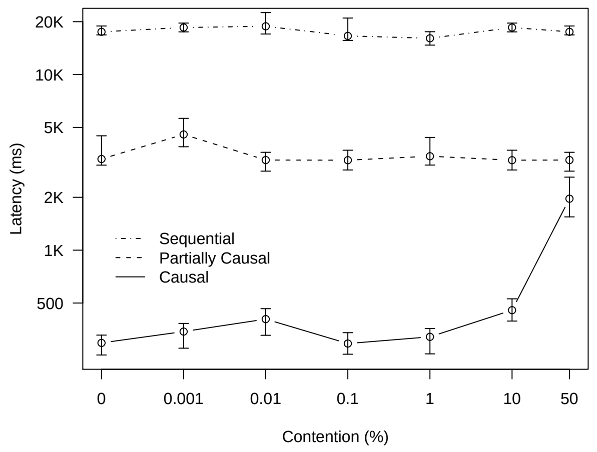


Figure 3: This log-log plot shows the median latency of enqueue operations as function of their collision likelihood. Various different consistency strategies are compared. Error bars are 80th and 20th percentile latency.

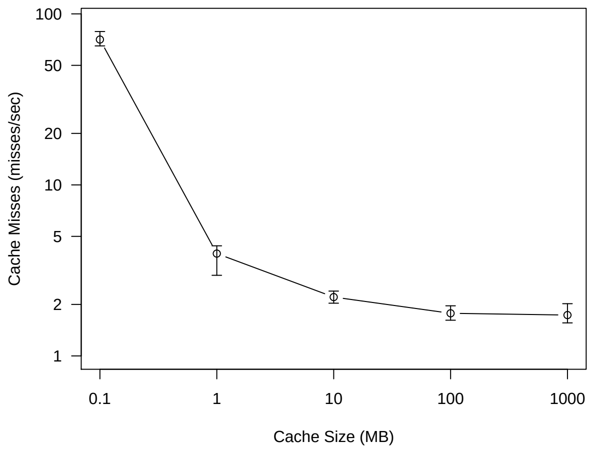


Figure 4: This log-log plot shows the median number of Bigtable read/sec as a function of TaskMaster's cache size.

for it's cache unless otherwise noted. The Bigtable cluster backing the TaskMaster contained 20 tablet servers allocated sufficient memory. The TaskMaster, Bigtable cluster, and all of the worker nodes each ran on separate machines in the same datacenter. As a result, network round-trip time between any worker and the TaskMaster was less than a millisecond. Network bandwidth was more than sufficient as each task's data payload was only 1KB. Each machine had two dual-core Opteron 2GHz chips and sufficient physical memory for their respective processes.

6.1 Effect of Optimizations

We have suggested a number of optimization to TaskMaster to improve performance. Each of these are evaluated using benchmarks specific to those optimization to tease out the effects on system throughput and latency.

Causal Consistency

The consistency benchmark evaluates the performance of causal consistency in comparison to strict sequential consistency. The likelihood of colliding *(queue, PID)* pairs is varied and the average operation latency is measured. The upper bound on update operation latency in the causal consistency model is shown by evaluating partially causal consistency which considers any updates operating on the same queue to be causal. In this test, we used 10 workers with 50 threads per worker, each issuing updates to a single TaskMaster.

Enforcing strict sequential consistency causes significant update latency for any contention value, as illustrated by Figure 3, since this consistency model considers all operations to collide. Partially causal consistency reduces update latency by more than 4-fold as each of the 10 workers are allowed to proceed in parallel since they update non-overlapping queues. There still exists contention amongst threads within a given worker causing all updates from a given worker to collide. True causal consistency provides another near 10-fold improvement in latency for small values of contention and is bounded above by partially causal consistency as expected. Sources of jitter in latency, as shown by the error bars, were caused by GFS hiccups propagated through Bigtable as increased mutation latency.

Caching

The cache benchmark measures frequency of cache misses which in turn result in Bigtable read operations. Leases are acquired in batches of 50 from 500 queues using 50 worker processes with 10 threads each. The cache size is varied and TaskMaster adaptively varies the number of prefetched tasks. The number of cache misses dramatically decreases once the cache is of any reasonable size, as shown by Figure 4. Increasing cache size beyond 10MB under this load yields diminishing returns because the cache utilization decreases. For this test, only a 47MB cache is required since tasks are 100 bytes and approximately 1000 tasks are cached with each refill of the 500 queues. As expected, the miss rate does not drop any further after the cache reaches this size. The cache miss rate never drops all the way to zero because queues occasionally require refilling.

Experiment	tasks/sec	variance
enqueue	1630	13.6%
FIFO enqueue	2386	8.7%
lease + dequeue	1485	6.8%

Table 1: The number of single-task operations per second enqueued, leased, and dequeued in aggregate across all workers.

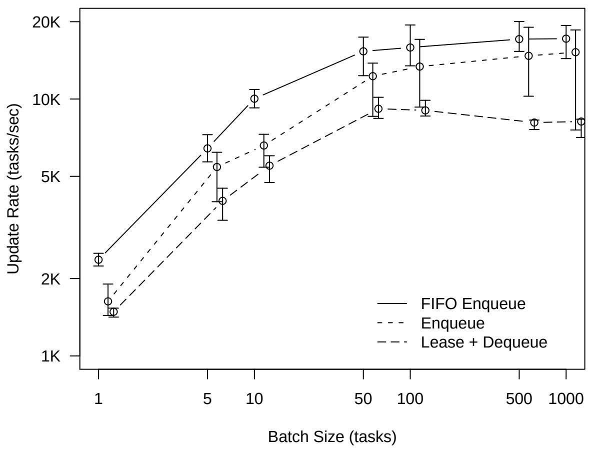
In practice, it is common for TaskMaster to be allocated upwards of 2GB of memory for caching. Typical TaskMaster instances have significantly more active queues, so a larger cache reduces the number of premature evictions. At any given time, these systems use more than 90% of their alloted memory resources. Limiting these evictions has a direct impact on latency of lease operations as they won't have to perform a Bigtable read to acquire tasks.

Single Node Throughput

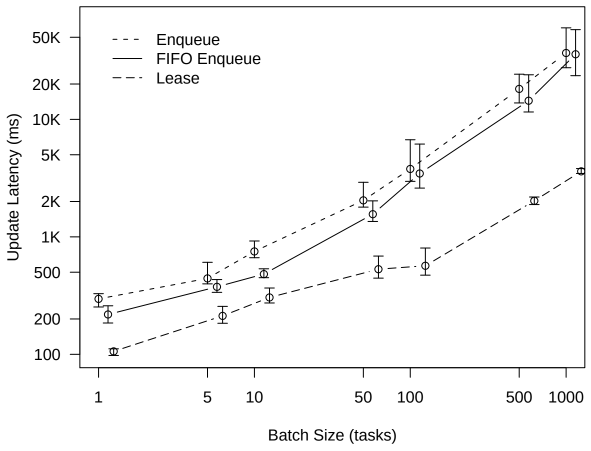
The throughput benchmark measures how many single task updates per second can be applied to a single TaskMaster node. This is a common use-case for latency sensitive applications which can't afford to delay operations while accumulating tasks to batch together. This benchmark uses 50 workers with 10 threads each. Table 1 shows that FIFO enqueues are the highest throughput operation as they only require a single Bigtable write. We don't require a read in this case because the PIDs are known to be unique. Recall, normal enqueues require a read and a write, FIFO enqueues only require a write, leases require a scan only when refilling the cache, and dequeues require a read when the previously leased task was evicted from the cache and a write.

We also see that lease and dequeue operations (performed in two sequential RPCs) have a similar throughput with respect to an enqueue operation (only a single RPC). This is explained by the fact that Bigtable random reads are slower than writes. Typically, lease and dequeue operations would be separated by some expensive processing step which we ignore because we are simply testing system throughput. Additionally, there is a fixed number of tasks which can be leased from a queue at once. This is the same user-defined value for the maximum refill size which was previously used to prevent cache thrashing. As a result, we must pair lease and dequeue operations for this benchmark to make progress.

The large variance for the enqueue operation can be explained by GFS hiccups propagated through Bigtable and exacerbated by having to commit log each operation before execution. Reads are more susceptible to these sort of irregularities since a read requires Bigtable to read part of an SSTable from GFS. In contrast, writes need only write to the Bigtable commit log which has mechanisms in place to lessen the impact of this exact event.



(a) Bulk Operations Rate



(b) Bulk Operations Latency

Figure 5: These log-log plots show the effect of bulk operations on 5(a) throughput and 5(b) and latency when varying the batch size. Points are artificially offset for clarity.

Bulk Operations

TaskMaster supports the batching of multiple updates into a single request to increase the throughput of the system. This reduces the per task overhead by amortizing the network round-trip, RPC parsing, and commit log delay incurred by each operation over all the tasks contained in that RPC. TaskMaster also allows lease requests to lease multiple tasks from multiple queues at once. This facility makes interacting with large quantities of queues more efficient.

For this test, we again used 50 processes with 10 threads each. Depending on the mode, they either enqueued work or leased and subsequently dequeued work from 10 distinct queues. Figure 5(a) shows that as we increase the batch size, throughput for each type of operation increases up to some threshold. The maximum throughput is bounded by the average mutation latency and the number of threads in TaskMaster. Increasing the number of TaskMaster threads has diminishing returns as context switching between threads begins to leach performance.

We can see that FIFO creates exhibit the highest throughput and lowest jitter because they only require a write. Lowest throughput is the lease and subsequent dequeue because those operations requires two separate RPCs and subsequently two commit logs and two Bigtable writes.

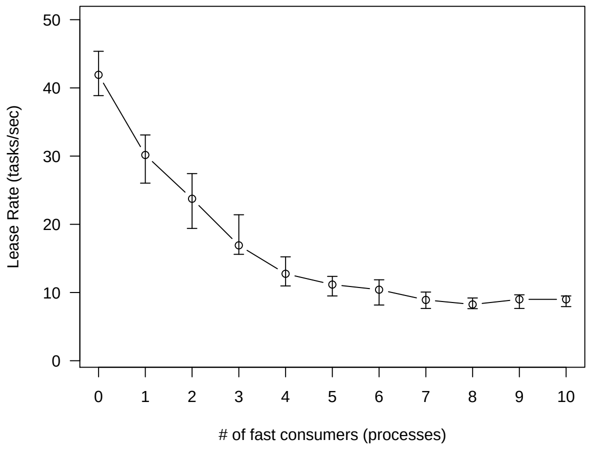
The latency for enqueue operations is elevated relative to lease, as seen in Figure 5(b), due to the relative slowness of random reads in Bigtable. This effect is also visible in the significant variance in enqueue throughput. We only measure the latency of lease operations as the subsequent dequeue operation exhibits the same latency properties as the enqueue operation. Lease operations are expected to be fast as they are serviced largely from the cache and only require a commit log write. The variance is caused by occasional cache refills.

6.2 Queue Isolation

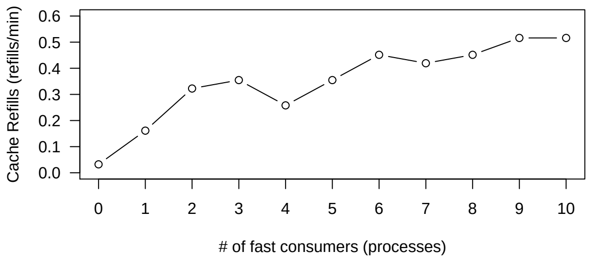
TaskMaster strives to ensure fairness of resources allocated to queues and the liveness of a worker's operations. To measure this, we induce load on the TaskMaster by having up to 10 workers, each with 50 threads, issuing updates containing 50 tasks to 10 distinct queues. We then have a lone single-threaded worker dequeuing tasks from a previously filled queue. We limit the cache's size to 10MB to put added pressure on the cache. As we can see in Figure 6(a), the single worker's throughput is significantly effected by the first few loading workers, but as more of them are added, there is a diminished effect. After the addition of the sixth, the single-threaded worker's performance doesn't degrade any further.

This effect on performance is caused mostly by increased write latency. As more loading workers are added, the number of total operations per second processed by TaskMaster increases. Similar to bulk operations, the maximum throughput is bounded causing latency to increase. Once this bound is reached, performance doesn't degrade any further.

We also measure the number of cache refills per minute for the relatively inactive queue to determine if the queue is being allocated a sufficient amount of the cache's memory. We can't directly measure the number of tasks queued because that value fluctuates as tasks are



(a) Effect on dequeuing throughput



(b) Effect on dequeuing queue's cache refills

Figure 6: These plots show the effect on 6(a) throughput and 6(b) cache refills that a varying number of 50-threaded enqueuing processes has on a single-threaded dequeuing process and associated queue.

dequeued and the cache is refilled. Figure 6(b) shows that the queue's cache refill rate does not increase significantly in the face of additional load on the TaskMaster. This confirms that TaskMaster is fairly distributing cache resources amongst the queues even with a large discrepancy in operation rate.

6.3 Cluster Performance

To measure performance of TaskMaster clusters, we vary the number of running TaskMaster nodes in a given cluster and measure aggregate throughput. Each TaskMaster, has 50 worker processes with 10 threads each producing load. In this case, we use a batch size of 50 tasks per operation. This was shown to have good throughput and latency properties by Figure 5. A single Bigtable cell consisting of 20 tablet servers held the Bigtables associated with the TaskMaster nodes.

We see in Figure 7 that aggregate throughput of random and FIFO enqueues scales almost linearly with the

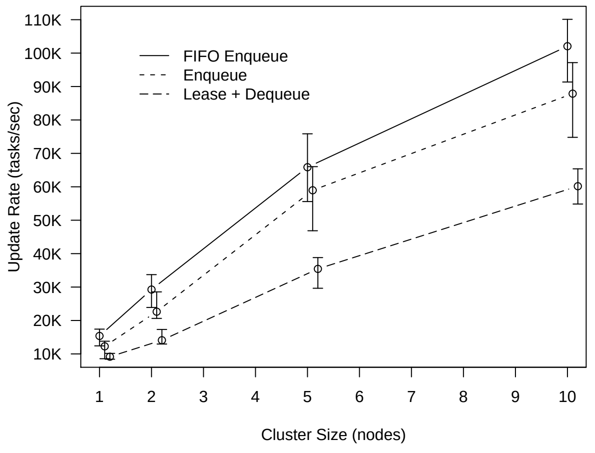


Figure 7: This plot shows the aggregate throughput of various operations as a function of cluster size. Points offset for clarity.

number of TaskMaster nodes. However, the aggregate throughput of lease and dequeue operations seems to be significantly sub-linear. Recall, we are performing a dequeue operation in addition to a lease operation. Since the latency of a lease operation is about half that of a enqueue/dequeue operation, we expect their combination to take 1.5 times longer than an enqueue. As a result, the throughput of enqueues is approximately 1.5 times higher than the throughput of lease and dequeue operations.

We also see the variance in throughput increasing somewhat sublinearly. These beneficial effects come from Bigtable latency spikes non-uniformly effecting the TaskMaster nodes. Thus, the detrimental effect of these spikes are essentially amortized over the cluster.

7 Example Applications

TaskMaster was originally designed for a structured data extraction application. The system consists of a number of stages which serve to fetch the documents, extract the structured data and links, uniqify and canonicalize links, and finally push the extracted data into an index. The new unique links found on a given page are fed back into the fetching stage and the cycle continues. In this application there are over 100 worker nodes and TaskMaster is managing *O(800M)* tasks across *O(20K)* queues.

Another application using TaskMaster measures the quality of ads served in response to queries. It is composed of a number of multi-stage FIFO pipelines and a set of queues used for coordination. First, a task is enqueued into the fetching coordination queue. A worker leases this task and enqueues a number of query tasks into the first pipeline where they are issued and post processed. Once that pipeline has drained, the task is dequeued from the query queue and a task is enqueued into the process queue atomically. This triggers further aggregate post processing of the queries results, and upon completion enqueues a task into the loader's queue. Finally, the loader takes the processed information and stores it into various repositories for wide-spread consumption. A single TaskMaster manages many such complex flows as multiple query sets are executed in parallel.

Google Docs uses TaskMaster to decouple the user action of requesting a document conversion from the document conversion mechanism itself. User requests are enqueued in one of many running TaskMaster instances such that a single TaskMaster node failure doesn't prevent users from requesting a document conversion. A set of workers lease tasks and send them to a pool of document converters. Upon completion, the worker stores the result in the appropriate location and dequeues the task.

The task is dequeued and a new task is enqueued in a backoff queue should a conversion fail for any reason. Exponential backoff is modeled by having multiple backoff queues, each representing some number of sequential conversion failures. Tasks enqueued in these backoff queues are given a PID representing an exponentially increasing time in the future dependent on the number of sequential conversion failures. Workers attempt to acquire leases on tasks whose PID is at most the current time so that these tasks are not leased until the backoff time expires.

In rare cases, a request is denied after some number of sequential failures. By bounding the maximum time a request remains in the system, this mechanism protects against requests that inadvertently trigger a bug that causes a failure in the conversion application. This event would be captured by the task's worker and would prevent future replays.

8 Related Work

TaskMaster's data model is similar to the transactional model proposed by [9]. They suggest the decomposition of long duration activities, like accounting for a day of purchase orders or crawling a website, into many inherently serial subactivities which can be run in parallel on multiple machines. The TaskMaster model takes this decomposition a step further, suggesting that these individual subactivities can be further partitioned into a sequence of stages to simplify application construction and provide better system transparency.

Linda [14], and more recently JavaSpaces [22] and TSpaces [21], suggest the use of unstructured tuple spaces a means of distributed communication and coordination. Instead of providing an arbitrary tuple store, TaskMaster imposes a priority-queue structure on this model providing an explicit order amongst tuples. Thus, tuples retrieved by workers are the highest priority tasks in a given queue rather than arbitrary tuples identified by their structure. Like tuple spaces, producers and consumers of given queues are decoupled both in time and space. TaskMaster also provides exclusive leases, stronger than the non-destructive `read()` primitive, as an additional coordination mechanism to prevent multiple workers from executing the same task.

Message Oriented Middleware [3] uses messages as the mechanism to allow existing application to easily communicate and coordinate. They suggest the use of messages as the glue to connect such systems with minimal modification to the existing application's design. TaskMaster is an instantiation of a Message Oriented Middleware system. In TaskMaster-based applications, tasks are the analog to messages as the glue to connect stages of processing pipeline together. Unlike messages, however, tasks are expected to be units of computation as opposed to event notification. Additionally, tasks have an order as defined by PIDs instead of the simple FIFO order given to messages.

SEDA [28] decomposes server-side processing of users requests into multiple stages. It uses queues to coordinate parallel execution of these stages in multiple server threads. TaskMaster extrapolated this mechanism to multiple processes, using queues to connect stages of processing. SEDA applications, such as the described HTTP server and a packet router, typically have real-time latency constraints. In contrast, typical TaskMaster applications only have strict latency requirements for enqueue operations, other operations have more flexible latency constraints.

TaskMaster is slight generalization of various publish/subscribe messaging systems surveyed in [13]. Both publish/subscribe and TaskMaster provide a means of process communication and coordination. However, publish/subscribe systems are aware of the set of publishers and subscribers and actively push message from publishers to the set of subscribers. TaskMaster has no knowledge of the produces and consumers of tasks, simplifying its construction and allowing for more workers. Consequently, TaskMaster tolerates consumer failure more gracefully; it simply leaves tasks enqueued until an interested worker dequeues them. Many publish/subscribe implementations have a fixed period during which they try to notify subscribers about the presence of a message. If some subscribers don't respond, messages may be dropped.

TaskMaster supports many queues as it anticipates applications to have many parallel pipelines, each consisting of several queues. In contrast, publish/subscribe systems expect that messages are published to a relatively small number of channels or topics. A publish/subscribe system could, in fact, be built using TaskMaster. Each channel would be associated with one or more TaskMaster queues. Publishers would enqueue messages into those queues and a pool of workers would be responsible for dequeuing and disseminating those messages to subscribers.

Several commercial software vendors have developed queuing systems similar to TaskMaster. The Java Community Process program defined the Java Message Service interface [16], a generic interface to such systems which provide both FIFO queuing and publish/subscribe mechanisms. WebSphere MQ [8], previously known as MQSeries [5], Oracle Advanced Queuing [25], and Sun's Java System Message Queue [23] all implement this interface and have varying scaling, security, and queue/channel administration properties. Security, namely queue access controls, is not a goal of TaskMaster, and administration of TaskMaster is simplified by ephemeral queues and efficient queue deletion.

DECmessageQ [7] provides multi-reader queues (MRQs) which either use a fixed set or priorities or FIFO for ordering. Amazon SQS [2] is the simplest system, only providing approximate FIFO ordering semantics. Also, messages in SQS have a fixed lifetime of 4 days because it is a shared service. None of these commercial systems provide the flexibility of free-form user specified PIDs nor to they provide leases to help tolerate consumer failure.

A sequences of MapReductions [10] could be used to build applications similar to those enabled by TaskMaster. The benefit of using TaskMaster is the parallelism of the different processing stages (here modeled as different MapReductions). By allowing all stages to process tasks at once, the latency of the logical computation (from the first enqueue until the final dequeue) is significantly reduced.

9 Conclusion

We have described the design and implementation of TaskMaster, a reliable queuing system used to simplify distributed system development. We have illustrated the power of PIDs to create meaningful queue orderings. Task leasing was shown to help applications reduce work duplication and better tolerate machine failure. We explained how atomic queue operations combined with causal consistency allow users to easily reason about TaskMaster without sacrificing performance. We described how to fairly distribute cache resources among a number of queues and how to limit data store reads by caching a contiguous range of tasks. Finally, we showed how simple queue statistics, like enqueue and dequeue rate, can help direct application debugging and optimization.

The versatility of TaskMaster's model is shown by its use in production over the last year for a variety of disparate applications. Our users appreciate the simplicity of the interface and the transparency provided by the queue statistics. TaskMaster is an excellent tool for building distributed systems, and we anticipate its continued use in the years to come.

Acknowledgments

The TaskMaster system and this paper has benefited from the helpful feedback of John Fisher-Ogden and Steve Gerding. Greg Billock, Caitlin Lovett, Tom Schoellhammer, and Joe White commented on earlier drafts of this paper. We also would like to thank Eddie Kohler for pointing out various helpful refinements to the system.

References

[1] Marcos Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. *Symposium on Operating Systems Principles*, 2007.

[2] Amazon. Simple Queue Service, 2008. `http://developer.amazonwebservices.com/connect/entry.jspa?externalID=1214`.

[3] Guruduth Banavar, Tushar Chandra, Robert Strom, and Daniel Sturman. A Case for Message Oriented Middleware. *Symposium on Distributed Computing*, 1999.

[4] Luiz A. Barroso, Jeffery Dean, and Urz Hölzle. Web Search for a Planet: The Google Cluster Architecture. *IEEE Computer Society*, 2003.

[5] Burnie Blakeley, Harry Harris, and Rhys Lewis. *Messaging and queueing using the MQI.* McGraw-Hill, Inc., New York, NY, USA, 1995.

[6] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *Symposium on Operating Systems Design and Implementation*, 2006.

[7] Digital Equiptment Corperation. DECMessageQ: Introduction to Message Queueing, 1994.

[8] Saida Davies and Peter Broadhurst. WebSphere MQ Fundamentals, 2005.

[9] Umeshwar Dayal, Meichun Hsu, and Rivka Ladin. A Transactional Model for Long-Running Activities. *Conference on Very Large Data Bases*, 1991.

[10] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Symposium on Operating Systems Design and Implementation*, 2004.

[11] Guiseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *Symposium on Operating Systems Principles*, 2007.

[12] Edsgra W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1(2):115–138, 1971.

[13] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computer Surveys*, 35(2):114–131, 2003.

[14] David Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[15] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. *Symposium on Operating Systems Principles*, 37(5):29–43, 2003.

[16] Mark Hapner, Rich Burridge, Rahul Sharma, Joseph Fialli, and Kate Stout. Java Message Service Specification, 2002.

[17] Phillip W. Hutto and Muslaque Ahamad. Slow memory: weakening consistency to enhance concurrency indistributed shared memories. *Conference on Distributed Computing Systems*, 1990.

[18] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. In *Symposium on Theory of Computing*, pages 654–663, New York, NY, USA, 1997. ACM.

[19] Leslie Lamport. Concurrent reading and writing. *Commun. ACM*, 20(11):806–811, 1977.

[20] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[21] Tobin J. Lehman, Alex Cozzi, Yuhong Xiong, Jonathan Gottschalk, Venu Vasudevan, Sean Landis, Pace Davis, Bruce Khavar, and Paul Bowman. Hitting the distributed computing sweet spot with TSpaces. *Computer Networks*, 35(4):457–472, 2001.

[22] Sun Microsystems. JavaSpaces Service Specification, 2007.

[23] Sun Microsystems. Sun Java System Message Queue, 2008. `http://www.sun.com/download/products.xml?id=41f9964d`.

[24] MySQL, 2008. `http://www.mysql.com`.

[25] Oracle. Oracle9i Application Developer's Guide - Advanced Queuing, 2002.

[26] Yasushi Saito, Brian N. Bershad, and Henry M. Levy. Manageability, availability and performance in Porcupine: a highly scalable, cluster-based mail service. *Symposium on Operating Systems Principles*, 1999.

[27] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM*, 31(4):149–160, 2001.

[28] Matt Welsh., David Culler, and Eric Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. *Symposium on Operating Systems Principles*, 2001.