acmqueue

Parallel Processing with Promises

A simple method of writing a collaborative system

Spencer Rathbun

In today's world, there are many reasons to write concurrent software. The desire to improve performance and increase throughput has led to many different asynchronous techniques. The techniques involved, however, are generally complex and the source of many subtle bugs, especially if they require shared mutable state. If shared state is not required, then these problems can be solved with a better abstraction called `promises` (https://en.wikipedia.org/wiki/Futures_and_promises). These allow programmers to hook asynchronous function calls together, waiting for each to return success or failure before running the next appropriate function in the chain.

Using the design pattern described in this article, programmers can build collaborative systems of threads or processes that are simple and easy to reason about because of the abstraction provided by `promises`. As `promises` are available in many languages, this design pattern can be implemented in whichever language is most appropriate or even in multiple languages for each process.

In the basic design, each piece of work requested of a worker creates a `promise` for that work. Dependent work hooks onto this `promise` and waits for it to complete before starting. If the work gets requested again, the new worker simply hooks onto the current `promise` chain and waits for the results with everything else. In short, the `promises` abstract the locking and allow the programmer to concentrate on the overlying pattern. By limiting the set of possible combinations the system can reach, handling all possible situations becomes simple, obvious, and removes the largest class of errors inherent to concurrent programs.

Developers often need to distribute work across a cluster of machines, processes, or threads. These distributed systems usually require complex code to make sure only one worker does each piece of work and that no work is missed, but it only gets worse when some of the work involved depends on previous pieces of work. How do these pieces synchronize themselves? When does a worker wait on another doing the work, and when does it take the work for itself?

This article shows how these and other problems easily resolve themselves without requiring a complex system of management, but instead use `promises` to create a state machine that simplifies the situation immensely. Unlike many distributed-programming algorithms, these workers collaborate to make sure only one of them does a unique unit of work and communicates with those workers that need that data. This eliminates the need for worker management, scales to an arbitrarily large cluster, and, most importantly, makes it easy to write and reason about the code.

What kind of work can be done in such a system? The answer, as it happens, is any work that can be uniquely named. Using a simple data structure to keep the name of the work with the associated `promise` keeps any outstanding work from being reprocessed and allows new requests to hook onto the outstanding promise. This simply and easily coordinates among workers and presents a simple API for programmers in whichever language they prefer, so long as it supports promises. The API is especially useful in server software because the work could be request processing for a Web server,

function computation for a MapReduce algorithm, or database queries.

Whatever is involved is unimportant to the system itself, so long as all of the workers use the same naming scheme. The simplest solution is to hash each piece of work to get this unique name. This is effective for most types of work, because they are easily represented as text and hashed.

When a worker receives one of these pieces of work, it requests a `promise` from a central authority within the system. As the maintainer of these promises, the central authority abstracts the complexity from the workers. This makes for an easy interface for programmers of the system and provides a single place to add a distributed-locking algorithm, if one is required. Once the worker receives the `promise`, it checks if it should begin processing, based on which end of the `promise` it received. Otherwise, it simply hooks up the appropriate function to resolve upon completion of the `promise` it received. Once the worker actually processing the request finishes, it sends the data down the `promise`. This notifies all the other workers about the results and allows them to process their completion handlers in parallel. This system lets programmers easily write concurrent processing systems without getting bogged down in the details of locking or complex lockless algorithms.

On a larger scale, if the system design calls for multiple servers or processes, the central authorities in each process must implement some kind of synchronization among themselves. Programmers must take care to determine where they fall in the CAP spectrum of such a distributed system. The `promises` do not, and cannot, solve the distributed consensus problem and, indeed, do not even know such coordination is happening. The best solution is to abstract the problem to an outside source designed specifically for it. If the work can be completed multiple times without detriment, then an available and partition-tolerant algorithm would be most appropriate. As much as possible, no worker will duplicate effort, but no disaster will occur in the edge case. For work that can happen only once, a consistent and partition-tolerant algorithm should be used instead for the locking mechanism. Some workers will not be able to process in the case of a network partition, but that is the tradeoff for consistency.

Upon completion of this system, programmers no longer need to worry about the details of how to synchronize. They can expend their effort instead on the problem they actually need to solve.

WHAT ARE PROMISES?

Since `promises` provide the core capabilities used by this algorithm, it is important to understand them. Fundamentally, a `promise` has two components, herein called `deferreds` and `futures`. A `deferred` is the input side of a `promise`, the piece where work actually happens. Once the work is accomplished, the `deferred` is resolved with the result. If an error occurs, the `deferred` can be rejected with the error instead. In either case, the `future` is the output side of a `promise`, which receives the result. `Futures` can be chained, so one `future` can act as the `deferred` for other `futures`.

Every `future` takes two functions: success and failure. If the `deferred` is resolved, then the success function is called with the resulting value. If the `deferred` is rejected, then the failure function is called with the error instead. If a `future` does not have the appropriate function defined, the value bubbles up to any attached `futures`.

Because `futures` can have an error function attached, they do not need to have error-handling code inside the success function. This avoids one of the problems of callbacks: mixing error handling with successful processing code. But what if the error is recoverable? Then there is an intermediate `future` with only an error function. If an error occurs, it can fix the error and then resolve itself. This

calls the next `future`'s success function. If no error occurs, the intermediate function simply passes the value to the next `future`'s success function. In either case, the appropriate processing occurs. This allows the `promise` to work as an asynchronous equivalent to a try/catch block.

`Promises` also have another feature, unlike their cousin, the callback: a `deferred` can be resolved or rejected with a `future`. When this happens, a new `promise` is inserted into the chain before any `futures` waiting on the current step. An important use of this feature is recovery code in a failure function. By resolving it with a `future`, a failed attempt to get data can try again without the other `futures` waiting on the data knowing that an error occurred. By using this technique, developers can encapsulate portions of a programming task into loosely coupled atomic functions whose ordering can easily be reasoned about, as shown in figure 1.

STATES

Because `promises` must run either their success function or failure function, the overall system maintained by the central authority has only a few states that must be reasoned about. First, there is the waiting `future`. Once a worker begins processing, the `future` could be left in a waiting state forever, since its associated `deferred` has not been resolved. To deal with this, the central authority

FIGURE 1

Promise Insertion

```
// Pseudocode demonstrating future insertion for error recovery

var future = doSomeWork();

// error handler function

var retry = function(error) {
       console.log("Had an error: ", error);

       //get a new promise for the work
       var newFuture = doSomeWork();

       // attach error handler to new promise and
       // return chained set, inserting into the chain
       // actual code should not blindly retry, as that could
       // potentially cause an infinite loop
       return newFuture.catch(retry);
};

// attach error handler and then later work
// if an error occurs, the catch inserts the
// new future to try again
future.catch(retry).then(doMoreWork);
```

should expose a function to the worker with which it can receive periodic notifications as a health pulse. Even using these periodic notifications the system can still have loss, however. The central authority should sweep the currently outstanding work and reject any `promises` that have waited too long. This will trigger the retry code down the chain.

Once the `promise` is resolved with the results of the work, it must start running through the `future` chains attached to it. During any of these steps, an error could occur. Luckily, `promises` transform thrown exceptions into calls for the next catch function. Passing these errors up the chain means they will be properly handled in the catch functions or passed along to the external callers. This prevents work from silently failing, or completing in a bad state.

Finally, if the work concludes, everything has functioned as desired, and the system has processed the request. Because `promises` abstract out to these three states, each state can be handled in the code in an obvious manner. This prevents many of the subtle bugs that plague distributed systems.

BASIC ALGORITHM DESIGN

At its core, the collaboration algorithm follows five simple steps, as shown in figure 2.

1. The unit of work gets an agreed-upon name so that multiple requests for the same work get the same `future`. Without an agreed-upon naming scheme, no worker could know whether another was already processing that work.
2. The `deferred` is requested from the hash table of currently operating requests. If no `deferred` exists, one is created, added to the hash table, and returned to the requestor. If there is already a `deferred`, the associated `future` is returned. Changing what gets returned in each case allows the worker to know whether it should begin processing or simply wait.
3. A catch function is attached to the `future`. This catch function retries if an error occurs, running the original function. By doing so, the new attempt will catch and try again. This allows for infinite retries, or by tracking a count of retry attempts, the function can fail up the chain instead of recovering. Since every worker dealing with this `promise` attaches its own catch in parallel, they all retry on failure. Any one of them might win the right to try again, providing a built-in communications system between the involved workers.
4. The work to do on the input gets attached so processing can occur once the data has arrived. It should not matter if multiple workers do the work at the same time, since their `promises` complete in parallel. Additionally, by returning this `promise`, work can be chained on preceding pieces of work getting completed.
5. Finally, if the worker received the `deferred`, then it should resolve the `deferred` once it has

FIGURE 2 **Basic Cache Algorithm**

finished processing the work. If a failure occurs during processing, then the `deferred` with the error should be rejected, which will notify all workers they should retry.

What happens when the work consists of small composable pieces? Recursively requesting each piece of work means the `deferred` can be resolved by the `future` for the preceding piece. This chains the small pieces together, with a worker walking backwards down the chain of things it needs until it finds either the beginning or another worker it can take advantage of. As each `promise` is asynchronous, the worker no longer needs to keep hold of it, and as the `futures` get data and are ready to run, any worker can pick them up. If there is only one chain, it will necessarily process one step at a time, the same as if no system was surrounding it. The pieces cannot go faster because of multiple workers, as they depend on the previous pieces, but shared pieces or multiple requests can take advantage of the already-processing parts.

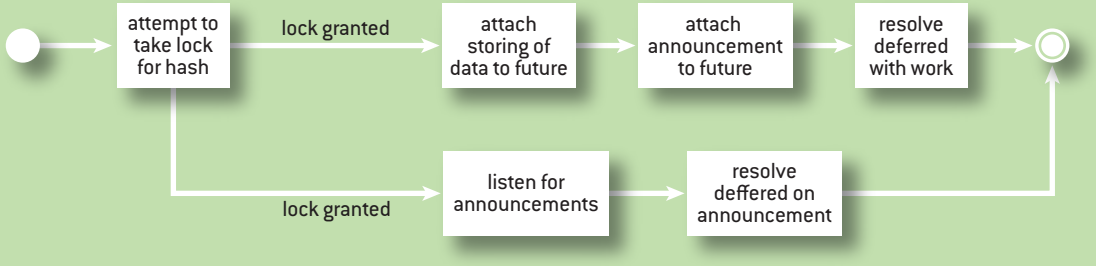
WHAT HAPPENS IN A DISTRIBUTED SYSTEM?

This basic algorithm requires a more complex central authority if the architecture precludes sharing a `promise` globally—for example, running on separate servers, or an application written in a language that cannot share `promises` across processes. In these cases a locking mechanism is required to keep the distributed central authority synchronized for a worker processing a `promise`. A Redis server or a chubby cluster can be used to provide this lock, since both systems are used for distributed locking. Care must be taken to determine that the chosen system provides the necessary consistency or availability. In addition, a method of broadcasting to the additional workers assists in speeding completion. Instead of waiting for them to check up on the lock, a message can be passed indicating that the work has been completed. After the internal `deferred` is received for the current process, a request must be made to the locking service.

If the locking service gives the lock to the requesting worker, it resolves the `deferred` as before. Another `promise` is added in parallel, however. On completion, this `promise` broadcasts a completion message to the distributed peers. The peers listen for such announcements and resolve their local `deferred` with the data by getting it either from a shared datastore or out of the broadcast message.

These steps must be added after the local `deferred` is requested. Since the local worker does not know whether any other worker has already started, it must attempt to take the lock whenever it starts. Failure to take the lock indicates some other worker has already started processing. This self-regulates the workers across nodes. The steps detailed in figure 3 show what happens.

FIGURE 3 **Cache Algorithm with Locking**



STAYING ALIVE

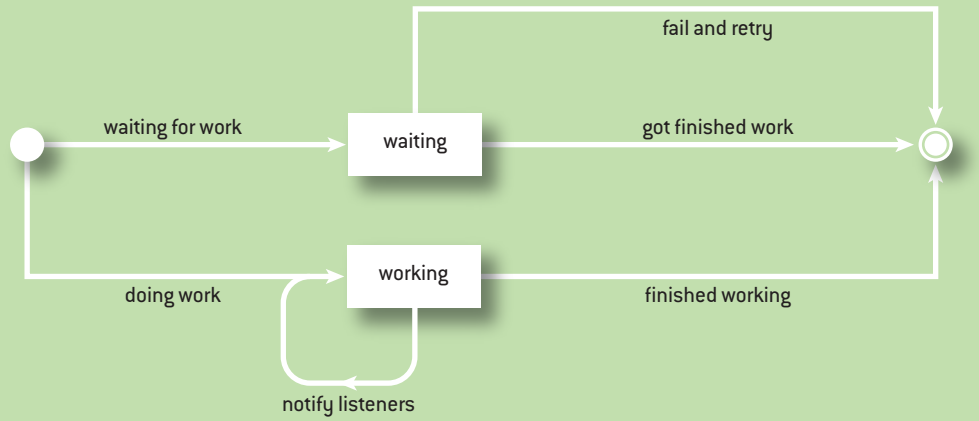
In any case, the worker processing the work needs to keep its peers updated on its progress. To that end, a periodic notification is sent from the worker to the central authority, acting as a health pulse. If this pulse is absent, then the central authority triggers the retry code for the other workers, recognizing the death of the one doing the work. Using an already-established code path to indicate failure means there are fewer states to reason about, and the workers can be certain that if a lock is out for work, someone is processing it or will be soon.

Additionally, `promises` protect the system from false death of a worker. If the worker fails to continue sending out the health pulse, another worker takes over the work, but the original worker continues processing the work and will probably complete before the original. The original `deferred`, however, has already been rejected by the missing health pulse, prompting another worker to take over. `Promises` can be completed only once, so when the original worker resolves the original `deferred`, nothing happens. All of the `promises` have had a new `promise` inserted into their chains—that is, they have all been told that the original worker is defunct and a new one is processing. No one needs to know or care whether the original worker has died or will complete its task eventually.

As figure 4 shows, the system has two main states: working and waiting. If a worker succeeds in getting the `deferred`, it enters the working state. While there, the worker must send out notifications to the central authority indicating it is still alive. If the system uses a locking service, then the notification also updates the lock time to indicate work is ongoing. This health pulse prevents other workers from waiting forever. Without it, a failed worker would hang the system forever.

Workers in the waiting state will get the completed work and finish, or else they will get rejected and run the retry code. If the central authority times out waiting for a worker's health pulse, then it checks with the locking service, if it exists. If there is no service or the lock time has failed to get an update, then the waiting workers have their `deferred` rejected with the retry error, causing all waiting workers to follow the basic error-recovery steps. A new worker will begin doing the work,

FIGURE 4 **Cache Notifier**



and if the original worker has not actually failed, it will resolve the previously rejected `promise`. Resolving or rejecting a completed `promise` is a `noop` (no operation) and gets ignored, as in figure 5.

IMPLEMENTATION EXPERIENCE

The design described here originally came out of the need to integrate a caching system for queries on top of a Mongo database cluster. Multiple Node.js processes spread across different servers would run queries for users simultaneously. Not only did the results need to be cached, but also large, long-running queries would lock the Mongo cluster if they were run many times in parallel. Since each query was a pipeline of steps to take, the queries could also build upon each other. A query matching certain documents, then sorting them, could have a grouping step added afterwards.

Naturally, recursing backwards through these steps and hashing the current segments provides a perfect naming scheme. If a user has already run the query, then the name can be looked up in the Redis database used to store the results and the query does not need to run. Otherwise, following the design described here, the appropriate `promise` is requested. If another user has already started the query, then no work needs to be done. This prevents overloading the Mongo database with the same queries continuously and gets the results back faster for multiple requests for the same query.

FIGURE 5

Sample Retry Code

```
// Pseudocode demonstrating future insertion for error recovery

var future = doSomeWork();

// error handler function

var retry = function(error) {
       console.log("Had an error: ", error);

       //get a new promise for the work
       var newFuture = doSomeWork();

       // attach error handler to new promise and
       // return chained set, inserting into the chain
       // actual code should not blindly retry, as that could
       // potentially cause an infinite loop
       return newFuture.catch(retry);
};

// attach error handler and then later work
// if an error occurs, the catch inserts the
// new future to try again
future.catch(retry).then(doMoreWork);
```

Working backwards through the set of query segments, the worker can hook onto any already-cached or running queries. Upon getting the results for an intermediate section, however, additional Mongo query work needs to happen. The mungedb-aggregate library (https://github.com/RiveraGroup/mungedb-aggregate) provides a one-to-one set of operators allowing the worker to finish any additional query segments. Without this library, the whole query would have had to run in Mongo, and running composable segments would have been infeasible. The only useful thing would be to cache the results, and there are already solutions for doing so.

The Node.js process has its own central authority to keep a single `promise` for any concurrent requests for a single query. This functioned well when a single process was in use for small deployments. Multiple concurrent requests would chain together, and they would transparently collaborate on the work. Unfortunately, the Node.js `promises` could not cross the process and server boundary. Upon scaling out to multiple servers, this necessitated the development of the fully distributed version of the system. Since Redis already had a place as the caching database, it made a natural abstraction for the distributed lock required. Redis was considered the best algorithm to use because it is available and partition-tolerant. If separate servers lost contact, then having each one run a single copy of the query was considered the right thing to do, as the results from the separate runs would be the same. The requestor always gets the results, and as much as possible the system minimizes the workload.

Once the implementation was started, the only major problem encountered was how to provide a health check for a currently processing worker. Luckily, the Q `promise` library (https://github.com/kriskowal/q) provides a notify function to its implementation. This provides a convenient hook for the central authority to listen on. The listener function updates the remote Node.js processes through the update time in the Redis database.

Throughout the implementation's life in production, there has not been a single unhandled error. Even with many queries hitting the system at once, and Node.js processes dying abruptly while processing, the system has continued to work as designed. Unlike complex locking code about which programmers have difficulty reasoning, the `promise` chains provide a clear and unambiguous way to coordinate the queries. Upon reading the code, a programmer can clearly see that it will work and which part is responsible for which task. Other systems may provide the same utility, but few are as easy to use and modify.

CONCLUSION

The system defined by these `promises` provides strong guarantees. Exceptions during resolution cause a rejection to flow through the `promise`, so work does not hang. `Promises` chain, so workers wait on previous work and then kick off to do their work on resolution. Notification listeners provide timeouts to prevent work from hanging. As few workers as possible will do the work, and the first one to succeed is the canonical answer provided to everyone. Most importantly, this algorithm provides a simple and correct method of writing a collaborative system.

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

SPENCER RATHBUN is a senior software architect for Rivera Group. He boxes for fun and enjoys learning about parsing theory and practice. His specialty is designing big-data solutions for a customer base that focuses primarily within the software architecture domain. To this end, he specializes in architecting creative solutions for real-world problems using structured/semi-structured data. Along the way he has lost his share of blood to the cutting edge.