Distributed web crawler using Python, Node.js, and MongoDB

William Schurman

Introduction

PyQA is a distributed web crawler and parser based on Celery, a distributed task queue written in Python. PyQA is composed of three non-blocking asynchronous tiers, allowing for non blocking API calls, increased throughput, and a more efficient distributed architecture.

Features

Distributed crawling and parsing via Celery

In most standard distributed system architectures, all distributed RPC requests and data pass through a master server or process. In PyQA, once a webpage has been crawled by a Celery worker, another Celery "subtask" is created and redistributed directly instead of sending the crawl data back to the central server. This greatly reduces the total I/O to the central system and automatically rebalances the system to evenly distribute work.

In the same way, if a worker becomes unresponsive, any important work that was being done will be redistributed to another worker without ever being routed through a central node. This makes worker failures transparent to the API layer.

Message passing via RabbitMQ

Instead of using periodic API polling to detect asynchronous state changes, messages are passed via RabbitMQ which allows for asynchronous message passing between components. In PyQA, messages are passed from API to client to signal a finished crawl or search. Messages are also used as the underlying system for Celery RPC.

Asynchronous design and threaded API

PyQA was built from the ground up to allow for asynchronous execution. This includes everything from the client request handler to the threaded crawl API and even database querying. The high level of threading and asynchronous design allows for a higher system throughput, from both a technical and user point of view.

Extensibility

PyQA is designed to enable simple implementation of additional parsers and search methods. Because Celery worker invocations are very similar to RPC calls, creating additional parsers is accomplished by extending the base parser that maintains the set of RPC constraints.

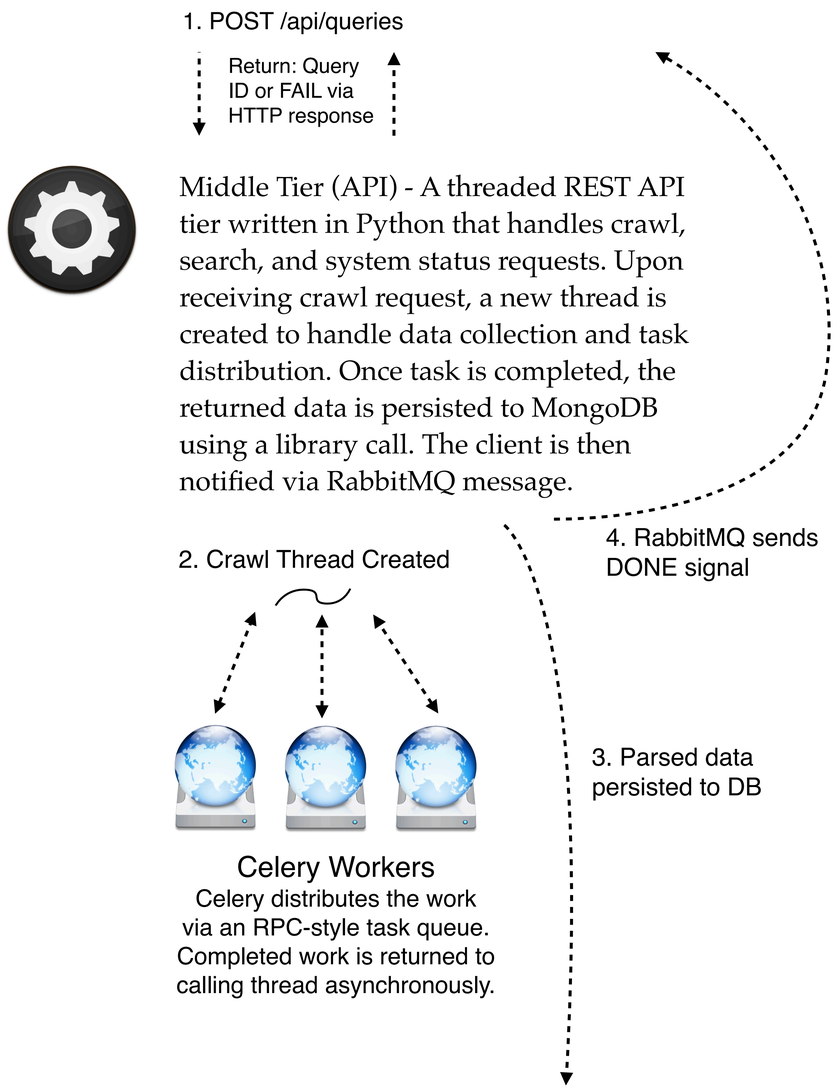
Searching

Search requests are handled in a way similar to crawl requests, although instead of being distributed to Celery workers, the requests are executed as MapReduce jobs in the inner tier (MongoDB).

Implementation

The implementation can be separated into a three tier cloud system:

Outer Tier (Client) - A stateless web frontend written in Node.js. User inputs a request via browser, websockets RPC transmits the request to node.js server instance which sends a request to the API tier.



Middle Tier (API) - A threaded REST API tier written in Python that handles crawl, search, and system status requests. Upon receiving crawl request, a new thread is created to handle data collection and task distribution. Once task is completed, the returned data is persisted to MongoDB using a library call. The client is then notified via RabbitMQ message.

Inner Tier (DB) - The innermost tier is a persistent MongoDB datastore and offline MapReduce computation layer.

Technologies

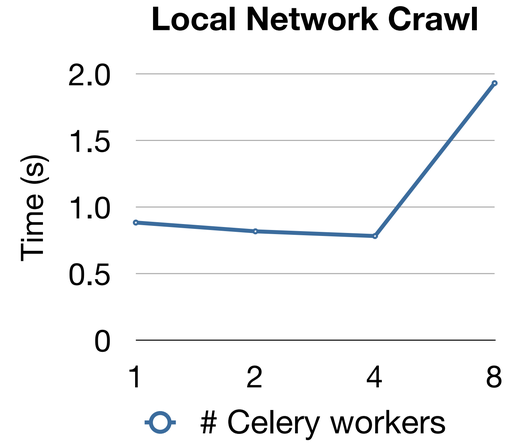
Node.js - An event-driven, non-blocking platform used to drive the frontend client server. (nodejs.org)
Celery - A asynchronous task queue based on distributed message passing. (celeryproject.org)
RabbitMQ - Message passing protocol based on AMQP. Used for all non-HTTP message passing between components. (www.rabbitmq.com)
MongoDB - Document-oriented NoSQL database. Also provides a built-in MapReduce implementation. (www.mongodb.org)

Results

The following tests analyzed the distributed capabilities of PyQA by recording the amount of time between the send of the first distributed crawl request and the reception of the final request's results.

The first test determined the efficiency of the system layout and the amount of latency added by further distribution in the ideal (no-latency) situation. The following test was run locally with all workers, scrapers, and parsers on the same local area network as the URLs being scraped.

Local Network Crawl



From these results, it can be determined that in a no-latency environment, further distributing the system adds unnecessary complication and causes the total time taken to increase.

On the other hand, in a high-latency environment where worker nodes are scattered in different locations, adding more worker nodes has the opposite effect.

Internet Crawl

For further information

Please contact *wts34@cornell.edu*. More information on this and related projects, as well as a PDF-version of this poster can be obtained at *github.com/wschurman*