Sprawl - A Distributed Data Processor

Kyle Knobloch
School of Computer Science
Carleton University
Ottawa Ontario, Canada

Quin Rider
School of Computer Science
Carleton University
Ottawa Ontario, Canada

The intent of this project is to develop a distributed data processor that would be easy to use and could manage a number of devices. Sprawl is modelled after BOINC [1], MapReduce [2] and Borg [3]. The goal was to develop a high performance "worker node" to run various binaries, as well as a server that would act as a user interface and scheduler.

***Index Terms*—Distributed, Data Processor, NodeJS, C, C++**

I. Introduction

Server sprawl is the situation in which server resources are not optimally used. Server sprawl can affect many different resources, including CPU, RAM and disk utilization. Many applications will fail to properly distribute components to fully utilize the available CPU resources, resulting in excess computational power that should instead be tasked with other jobs. Similarly, RAM and disk space may also be in excess on some servers while others are starved. Maximizing CPU usage is a fairly simple task, and can be done via semi-intelligent load balancing. RAM and disk sprawl, however, is not nearly as simplistic, as interacting with RAM across network connections is inherently difficult, and disk usage is best left to existing storage solutions. Due to the time limitations imposed on this project, only CPU sprawl will be tackled. This project focuses primarily on the distribution of workloads in which can be split into smaller workloads, making it better suited for "embarrassingly parallel workloads" than the distribution of application components.

II. Development

A. Design

A Sprawl cluster consists of two core components: An "Overlord" and one or more "Workers". An Overlord Server (OLS) is tasked with job distribution among the workers, as well as providing a user interface that allows clients to design and execute jobs. Workers are lightweight programs in which receive job requests and execute tasks. All communications between an Overlord and its workers is done via HTTP (with or without SSL encryption depending on the configuration). For simplicity, communications are encoded as JSON objects, allowing for easily parsed data transactions and with a highly extensible nature.

No funding was received for this project, however it will be used as a project grade for COMP 4000 A - F21 at Carleton University, Ottawa Ontario, Canada.

B. Worker

Workers were developed using C and C++. The choice of languages boils down to performance. Both C and C++ can be used to develop highly performing software, as this as a requirement for the Worker. The team has a strong background in these languages, so the pace of development was not significantly hindered by this choice.

A worker runs a lightweight HTTP server that accepts messages on a handful of endpoints. At creation, a worker will first attempt to load a configuration file from disk that specifies information about both it and the OLS it is to connect to. New workers POST information about themselves to their OLS, including the port in which they are running their API server on. The OLS mines the workers address from the POST headers, and retrieves the workers server port from the posted data. All communication (other than error codes and notice of termination) following this point is initiated by the OLS.

Workers exist in an "idle" state until the OLS POSTs work to their `/jobboard` endpoint. Once a job has been posted, the worker will then validate it by checking the jobs task binaries against its own known binaries and then performs "linting" on command arguments to ensure there are no extra commands hidden in them. This is done to avoid hostile execution unapproved binaries.

If a job runs to completion without any errors, a worker will post the stdout content of the last task to the OLS at `/jobs/return/:id`, then it returns to the *idle* state.

On the other hand, if a job is found to be invalid prior to execution, which occurs when the specified *runtime* is not available, or invalid arguments or binaries are specified, then the worker will respond to the job POST with a JSON encoded error message.

If a job fails during the execution of a task, then the worker posts the last tasks return code to */jobs/returncode/:id* and the stdout of the last task to */jobs/returnerr/:id* to allow user debugging of the issue.

Runtimes: In addition to the JSON configuration file required by a worker, runtime JSON files are also mandatory. These specify which binaries a worker is allowed to run, thus creating a somewhat lacklustre, but better than nothing safety surrounding the execution of tasks.

In summary, a worker listens for jobs, executes the tasks defined in jobs (possibly in parallel, if multiple threads are present on the system), then returns results to an OLS upon job completion or failure.

C. Overlord Server (OLS)

The Express.JS Framework was selected to develop a RESTful API that would be able to consumed by the worker as well as the end user.

Client Interface: Jobs are created through the graphical interface and have four major components: Name, Owner, input file and tasks. The Name and Owner are just to track who is responsible for the job and could be used in the future for more access control. The input file is always downloaded by the worker included in the first task in the task list. Of course binaries are able to ignore this input if needed. The tasks assigned to the job are executed in order and are made up of a binary and arguments where the last argument is `< {inputFile}` where it is replaced with the file locations locally.

There are also tables where you can view the tasks, jobs and workers on the server. These tables are generated client side by making a call to the API. Tasks are made up of a binary then arguments. The binary is typically only an approved binary that has been installed on the server. For some basic testing the primes binary was written that takes input from a file to determine of a range of numbers are prime or not.

Database & API: The database is currently a flat file JSON file that contains three top level key entries, `jobs`, `workers` and `tasks`. The top level keys contain an array of objects that correspond to those types. The array value for these keys is not the best option as deletion is not possible without changing the indexes, as such both objects have a status key that can be set to deleted. The API has three endpoints `/jobs`, `/workers` and `/tasks`. The jobs endpoint is used to submit and update jobs as they are completed. It is also where the worker returns the job to the overlord server. The workers endpoint is used to initialize and update the status of workers and is where workers update their own status if needed. The tasks endpoint is used to create command that the worker will execute for the jobs.

In Appendix A all of the endpoints and their functions are document. The most commonly used endpoints are `/workers/init`, `/jobs/add`, and `/jobs/return`. These are the endpoints that the client and workers typically access to communicate with the OLS.

Scheduler: The scheduler is designed to be executed every minute if there is a job that has yet to be assigned. When the scheduler function runs it checks all of the jobs to find the ones that has not been assigned to a worker yet, typically this is a 'Created' status. Once it has identified the jobs it then identifies all the workers that have a status if 'Init' or 'Idle'. From there it assigned the oldest job to the first worker it finds. It continues this process until either there are no more jobs or workers. If there remains jobs it will wait about one minute before running itself again. The scheduler is managed by the Cron npm package that will automatically execute the function on its schedule. This approach is a naive approach but actives the goal of being able to automatically assign jobs.

III. Design Choices

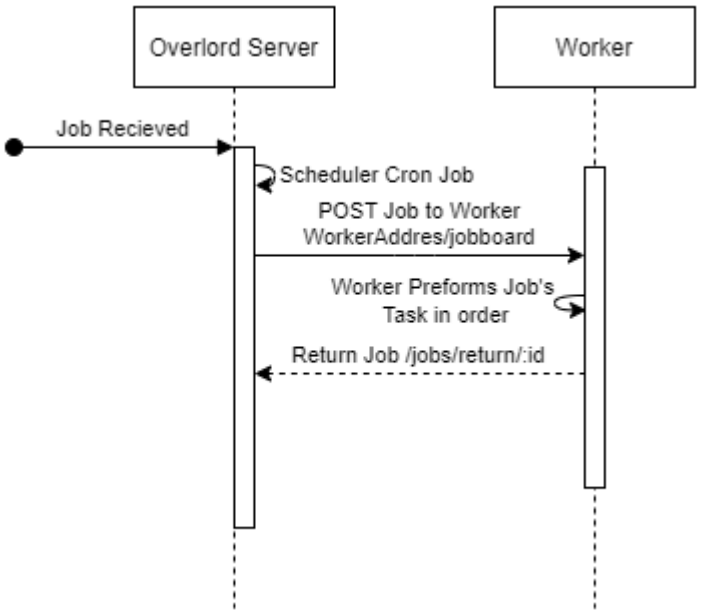

Fig. 1. Sequence Diagram for OLS receiving a Job from a client, send the job to the client and then receiving the job from the client.

Throughout this semester discussions around many different distributed technologies, and found most of them to have one thing in common: Linux. Linux is a good target platform for Sprawl, as it provides many niceties that ease the development of a system like Sprawl. A lightweight Linux distribution provides an environment in which Worker nodes can achieve maximal performance, and the plethora of libraries supported by modern Linux distributions made it trivial to find libraries where needed.

HTTP vs. HTTPS: Communications wrapped in HTTP would suffice. HTTP is a well defined standard, and Sprawl is meant for large workloads, thus the overhead of HTTP is not large enough to become an issue for performance (given sufficiently large workloads). While the OLS does support receiving HTTPS requests easily, the worker does not. The work in which the test cluster is intended to run doesn't require encryption, as nothing private or confidential is handled.

Security: Due to time constraints, the initial SSL requirements were abandoned and dropped from HTTPS to just HTTP. The groundwork for handling SSL is in place, and might actually work however, it would be best to focus on core functionality first and security later, thus this has not been enabled or tested.

Linux: Linux can be lightweight depending on the distribution, has a colossal set of packages, libraries and applications developed for it, and, best of all, is completely free. All three of these things supported choosing it as the base to build Sprawl atop of. Early development of the worker nodes was aimed at cross-platform compatibility, but this was quickly scrapped in favour of rapid development, and the introduction of WSL2 into windows, which made it trivial to run workers under Windows 10 and 11. Most worker development took place in Windows 11 using WSL2, and some was done using a Raspberry Pi 4B (mostly to play around with the surprisingly capable board, but also to help debug some ARM related issues and nuances that stopped things from compiling correctly).

A. Development Struggles

Struggling with time management and other things was one of the biggest obstacles. Particularly because of demanding course loads this semester. There was quite a large amount of work done to get to the state Sprawl is in now, however, the goals were over ambitious and naive to think it could be implemented in the dwindling time that remained in the semester. Many hours went into writing the underlying libraries required to create Sprawl, and are confident that with a lot more extra time the original goals would have been achieved.

One of the hardest things to test and debug was the communication between the worker and the OLS. As first the worker would reach out to the OLS and ask for its own status as needed. This approach is not as useful as it could overload the OLS and create issues down the line. This meant that when a job was assigned to a worker, the OLS would reach out to the worker to let it know. The worker would continue to tell the OLS if it was deleting itself and when a job completed. The debugging of the C/C++ worker and NodeJS making a simple TCP HTTP request had many road blocks. The first was due to the firewalls as ports needed to be opened to allow this in, the port `40401` was set to be the default port. Then, the firewall needed to be configured to ensure that it was allowing these TCP connections through. Then finally after many `netcat` commands that the OLS was sending it to the wrong URL at the worker! Once that was sorted it was a breeze to get the worker and OLS talk to each other.

IV. Hardware

A mixture of hardware was used to test the Sprawl system. Initially, the plan was to use some server-grade hardware to run things, but due to unforeseen circumstances, laptops and Raspberry Pi boards were used instead. Development was primarily done under Windows, with the Worker being mostly developed under WSL2 and later on a Pi 4B to help debug architectural issues (such as the requirement of additional libraries (-lm and -latomic), as well as different library versions requiring minor updates to be made to code. A hand-full of Raspberry Pi boards, ranging from v1.1 3B+ to the latest 4B+ models, and decided that they would suffice for showing of the key features of Sprawl. A Pi B+ board was used to host the Overlord Server, as it had the lowest specs and had access to a public IPv4 address, while a Pi 4B+ was used to test the cross-platform and architecture capabilities of workers.

Later, a two core Intel server was set up to do further testing of the Worker. In the demo provided in section VI-A the Intel server was running two workers and a Raspberry Pi B+ was running the OLS. The OLS does not require much computation power as it is a simple web server where the workers require much stronger power.

Development was done under both Windows and Linux distributions (varying), as well as different CPU architectures. Early builds of the workers were cross platform, however focus was on Linux to accelerate development.

V. Experience

A. Testing

Due to time limitations some testing which had been planned was scrapped. Instead, most of the testing went into ensuring the project was stable - that is to say, that it functioned correctly under load and yielded results or errors as expected.

The majority of testing of worker-OLS interactions took place in a single-worker environment, in which it execute the ”primes” binary on varying numerical ranges. Sticking to a one worker environment for interactions with the OLS due to unforeseen networking issues and firewall rules. This made it impossible to test the OLS against a large network of worker nodes, so a quick alternative to the OLS was developed to allow testing of multiple workers and all of the worker endpoints.

A lightweight ”CLIServer” was developed to fill-in as an OLS for testing, though it has limited functionality. The CLIServer merely accepts job requests, distributes the job, and stimulates all of the endpoints made available by a worker in order to execute the job, check it’s ”heartbeat” and obtain results. The CLIServer supports an arbitrary number of workers and clients, which allowed testing of multi-job scenarios without having the primary OLS portion of this project completed.

The ”primes” binary simply determines if each number in the given ranges are prime or not, and returns a newline-separated list of those that are. To test the multi-threaded nature of workers, a large range of fairly large numbers (e.g., 1000000-2000000) was used as a job. This provided enough time to post multiple jobs and fully saturate the available threads of the workers before the first completed, verifying that the workers could properly handle multiple jobs at a time (given the threads to do so) without stepping on each-other.

B. Kyle’s Experience

The development of this project was very informative and a great project to work on. Developing the OLS database and RESTful API was a good experience and built off of previous courses. The communication between the OLS and the Worker was the most interesting to work on. Ultimately it is two web servers talking to each other but it does the trick to

handle much of the backed that other distributed systems need. An alternative to the design choices here would be to have sockets that are constantly kept alive by the Worker and OLS communicating. This approach has draw backs as it would require much more bandwidth as well as an extreme load on the OLS when there are many workers.

Testing the OLS and running the prime jobs I created for the YouTube video showed clearly what was lacking. Using the system while not clunky left much to be desired by the user. The most notable was when a worker was Idle while jobs were sitting in the 'Received' state. Another limitation was that the OLS could only support files up to a few gigabytes in size due to the limitations of the Raspberry Pi's hardware as well as the Express.JS framework. This left much to be desired.

This is the first distributed system I have developed and it has been very informative. I can understand how the previous systems discussed in class like Plan 9 [4] and others were incredibly use full and successful but ultimately had downfalls. I would like to point out that this was developed from scratch with minimal experience with developing distributed systems. It is still a very impressive feat with the final product that was developed.

C. Quin's Experience

Development of the worker was slow in the beginning, primarily due to the number of prerequisite libraries developed and researched before proper progress could be made. In classical "me" fashion, I spent far too much time trying to make everything optimal at the expense of, well, time. From scratch, thread-pool, JSON and XML-like libraries were written for this project. Wrappers around existing lightweight HTTP server and client libraries were created for ease-of use, and then everything was glued together. The worker side of things turned out to be not nearly as complex as I had expected. The design choice of using HTTP meant communications were simple and easy to handle, and using JSON for messages kept everything well formed and easily modifiable whenever required. Memory leaks were a fairly rare occurrence, and it was far more common for me to stumble upon an invalid pointer, which was usually the result of freeing things without setting the variable to NULL (and thus easy to find using valgrind).

As Kyle has mentioned, communications ended up being quite the hassle when we tried setting up a cluster across multiple sites (his home and my own).

In our last-minute panic testing of workers with the OLS, we ran into issues with our "primes" binary failing to execute properly on Kyle's server due to a Segmentation Fault, which lead to some rather late debugging of something we did not even consider would become such a time consuming portion of this project.

In an attempt to display the performance of workers, I coded wrote a simple script to POST jobs to the CLIServer and have it distribute them efficiently. These tests lead to the discovery that running workers with more threads than that available actually increased run-time performance. My current thoughts on this are that this is making optimal use of the networking delays which are likely the greatest overhead of the current implementation. For workload with larger data sets, or longer executions times in general, I expect this to have a smaller impact on performance. Our demo video contains a segment showing the adjustment of the workerConf.json file to have a worker use 4 threads rather than 1; the workers throughput is visible greater than others.

Testing of the workers themselves proved simple enough, as they are not picky about their API and will simply run any job thrown their way. This made it easy to verify their ability to handle large workloads without memory leaks or inflation. The real struggle was testing the Sprawl cluster as a whole unit.

Ultimately, there was a lot more involved in this project than we anticipated. For better or worse, it made us test our debugging skills (more than we would like to admit). More time was spent making components work together rather than writing the components themselves, which jives with the early class readings from when distribution of components was still a fresh idea.

VI. Demos & Code

A. *YouTube Demo*

A demo is available at YouTube https://youtu.be/-ZiLc8fV2bU. This shows off a basic use case of Sprawl as well as an advanced use case that proves its potential.

B. Live Demo

A version of the software is likely still running at https://ddp.kyleknobloch.ca. NOTE: submitting a job is password protected to stop anyone submitting to overload the server or workers with malicious code.

C. GitHub

The code is currently private to avoid academic integrity violations. On request, code is available at GitHub https://github.com/ddp-comp4000-f21/.

VII. Future Work

There is a lot more that could be added to this project to make it a more robust system. As mentioned already, things on the user side like authentication and a better database would make this a much more usable system. Currently, the system cannot handle many jobs. Little additional time would be required to handle larger workloads; it is merely a matter of job distribution changes and updates to server-side logic.

Worker: Due to time constraints, the networking design of workers requires that outgoing messages be sequential, thus regardless of how many threads a worker is making available to run jobs, the results are set out one at a time. This would be a bottleneck in the situation that jobs are completing faster than it takes to send their results, and could be fixed by reworking the networking to perform threaded

operations. Another missing feature is ”sandboxing”. Opting for the simpler method of requiring verified binaries, however this is a serious limitation. A better solution would be to create a ”chroot jail” in which jobs would be run in, isolated from the rest of the system. This would allow users to safely execute various binaries without harming the underlying system. The technology used in this project is liable to be leveraged in future ones, as it provides a solid framework in which to build upon.

Security: A switch from HTTP to HTTPS for communications, along with additional data encryption methods for disk operations or local transfers is required in order for this system to become a reasonable choice in production environments. Password protection on the OLS stops hostile actors from using it to execute malicious code, but workers will blindly execute jobs when any valid job is posted to its */jobboard* endpoint, making it a potential target for denial-of-service attacks as it stands. Files that are kept around on an OLS should be encrypted for additional security (perhaps as an optional feature), and all user interactions should be performed over secure sockets. These features were not implemented as they were seen as ”niceties” rather than requirements for the purpose of this project, which was purely for the experience of developing a simple DDP.

Overlord Server: To keep things simple, initially the usage of simple incremented IDs for jobs and workers. It was highly unlikely that the IDs would ever perform an integer wrap and result in duplicated IDs, as this would require more than 2^{64} jobs or workers. In reality, however, large systems with high job throughput would require a more robust ID system. An improvement would be to implement UUIDs for both. This would allow for future expansion of Sprawl into multi-OLS environments while avoiding ID clashing.

The scheduler leaves much to be desired by the end user. In its current state it is a relatively simple scheduler that gets called every minute. A better option would be to use a lock file while it is running and call the function many times on specific events as well as on a regular interview. These events would be when a job is created and returned so that a Worker is never idle for long. The initial design had a concern for a race condition where it would over assign if called many times.

The OLS currently does not collect much information from the Worker when it is initialized. A late addition was the run times array which listed the available binaries. However this information was never integrated into the OLS when determining what jobs to send workers. A late development in the Worker also included the number of threads it could support. This development would mean that many jobs could be posted to the `/jobboard` endpoint on the worker but this was not planned for and as such was never used by the OLS.

Finally, the MapReduce functionality is the most glaring missing feature. This feature required a lot of development to get working and just did not fit into the time frame. The MapReduce feature would allow for really large jobs to be split into many, smaller jobs that would make it so that the user did not have to split up the input manually but instead the OLS would optimize it.

VIII. Conclusion

There were big plans in place for this project that would make it an impressive feat to complete in about 3 months time. This was way more work than anticipated. It was assumed that there would be enough time to complete the project within the remainder of the semester, and that it would be easy to do. It was not. A great deal was learned about about software design as well as how complicated distributed systems are. Objectively, this project is easy: Make a RESTful API, a worker and a Scheduler - it was getting those three systems to work together that proved rather difficult and time consuming. A lot was learned in this process and will most certainly be used in the future projects in our academic and professional careers.

Acknowledgements

Many thanks to Prof. Anil Somayaji for supporting this project, as well as Teaching Assistant William F. for helping us narrow down our targets to obtain a more reasonable goal (even though we still bit off more than we could chew).

Appendix A

A list of the Overlord Server API Endpoint and their documentation is provided below. This API is used extensively to control the OLS and worker and is accessed by both the client and the workers.

A. Jobs

- `/jobs` Returns all of the jobs data
- `/jobs/:id` Returns jobs data specified by `:id`
- `/jobs/add` Creates a new job from a POST request containing jobName, jobOwner, jobTasks, and inputFile
- `/jobs/delete/:id` Sets the jobs current status to deleted specified by `:id`
- `/jobs/set/:status/:id` Sets the jobs current status to `:status` specified by `:id`
- `/jobs/assignworker` Assigns the job to a worker specified by a POST request containing `id=JOBID&workerid=WORKERID`. This should only be used for manual assignment, the scheduler will typically assign jobs internally
- `/jobs/return/:id` Returns the job specified by `:id` data to the Overlord Server as MIME type `text/plain`
- `/jobs/returnerr/:id` Returns the job error output specified by job `:id` data to the Overlord Server as MIME type `text/plain`
- `/jobs/returncode/:id` Returns the job error exit code (non-zero) specified by job `:id` as MIME type `text/plain` but should be a number

B. Workers

- `/workers` Returns all of the workers data
- `/workers/:id` Returns workers data specified by `:id`
- `/workers/init` Initializes a worker with a POST request containing serverAddress and accepted run times, will return the worker's ID as a JSON object
- `/workers/delete/:id` Sets the workers current status to deleted specified by `:id`
- `/workers/set/:status/:id` Sets the workers current status to `:status` specified by `:id`

C. Tasks

- `/tasks` Returns all of the tasks data
- `/tasks/:id` Returns task data specified by `:id`
- `/tasks/add` Initializes a task with a request containing binary and args, will return the task's ID as a string
- `/tasks/delete/:id` Sets the tasks current status to deleted specified by `:id`

D. Admin

- `/admin/deletedatabase` Completely removes the database and sets it to its default empty values
- `/admin/deletedatabase/:db` Completely removes the table specified by `:db` and sets it to its default empty value

APPENDIX B

A list of the Worker HTTP server endpoints. This list excludes those which were not completed and/or tested.

- `/jobboard` Location to POST job JSON description, which will be parsed for validity. The worker will either notify the sender of success for failure with varying messages to help in debugging malformed job requests.
- `/kill` POST anything (preferably nothing) to this endpoint to terminate the worker. This does *not* require any kind of verification via passphrases, and simply notifies the worker that it must terminate after its current tasks have completed.

REFERENCES

[1] D. P. Anderson, "BOINC: a system for public-resource computing and storage," Fifth IEEE/ACM International Workshop on Grid Computing, 2004, pp. 4-10.

[2] J. Dean and S. Ghemawat "MapReduce: Simplified Data Processing on Large Clusters" in OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA (2004), pp. 137-150.

[3] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes "Large-scale cluster management at Google with Borg". In Proceedings of the Tenth European Conference on Computer Systems (EuroSys '15). Association for Computing Machinery, New York, NY, USA, 2015, Article 18, 1–17.

[4] D. Presotto, R. Pike, K. Thompson and H. Trickey "Plan 9, A Distributed System". In Proceedings of the Spring 1991 EurOpen Conference 1991, 43-50.