Limo and Pogo: Parallelizing task execution at Yahoo!

Michael S. Fischer <mfischer@yahoo-inc.com>
Nicholas Harteau <nrh@yahoo-inc.com>
Andrew Sloane <asloane@yahoo-inc.com>

Abstract

Because Yahoo! properties attract a wide audience, massive farms of computers are needed to serve content and provide services. These farms consist of many "clusters" of hosts that are similarly configured. In order to efficiently administer such large host groups, novel tools are needed. This paper discusses Limo, an agent-based service for performing tasks on arbitrary numbers of hosts in parallel; and Pogo, a service that expands on Limo to address redundancy and allows modeling of complex deployments.

The Problem

Yahoo! has the fortunate problem of being so popular that we cannot serve the content demanded by users or analyze our data with just a few servers. Many properties, including Yahoo! Mail, Finance and Search require tens of thousands of computers to meet demand, and Grid services may ultimately have an even larger scale. These servers require regular administration; fixing bugs and releasing new features requires that we install new code on them frequently.

When Limo was conceived in late 2007, there were few options for performing these duties in parallel to ensure that they could be completed in a reasonable timeframe. *yinst*, for instance, has a `-jobs` option, but its output is not collated, making it difficult for the service engineer to parse the output and debug issues that may have occurred. A service-based solution, *yinst_server* (a component of Taxi), has crippling performance issues and a baroque client interface. Common open-source tools such as *pssh*, *mussh*, etc, are available but nearly all run directly on the user's workstation, greatly limiting both their scalability and utility in a team environment.

Fundamental Principles

In designing a solution we adhered to the following principles:

1. **K.I.S.S.** Start with the simplest conceivable design that will solve the problem. Reject components that add complexity but are not essential, especially where adequate, lightweight and well-tested components can be provided by the operating system.
2. **Make it generic.** In keeping with the UNIX philosophy of building chains of small tools, each of which do one thing well, we intentionally architected Limo to be a generic task execution system. This ensures maximum flexibility and reduces the risk that unforeseen use cases will be difficult to implement.
3. **Make it scale.** A multi-threaded remote SSH client that runs on a single workstation has limited scalability. We needed a solution that could perform at Yahoo! scale – thousands of targets.
4. **Don't re-invent protocols.** Only develop your own protocol if there is a compelling reason to do so—and where you must, look to other well-known Internet protocols for inspiration.
5. **Don't re-implement the basics. Don't expand the web of trust.** Limo relies on preexisting, battle-tested authentication and authorization components. Tinkering with or replacing these would have added considerable administrative overhead and would have required extensive analysis to ensure the security of the system.
6. **Make the system easy to use and debug.** This speaks for itself.

Limo

Technology

Limo is built on a carefully chosen and diverse suite of components.

I/O interface: AnyEvent. AnyEvent is a powerful, event-driven I/O framework for Perl. It provides a unified, intuitive model for developing applications that use asynchronous network I/O and supports many different event loop backends - EV, libevent, select(2) and even POE. AnyEvent supports SSL encryption and authentication in both client and server mode and supports bidirectional certificate validation. AnyEvent also makes it easy to write timers and signal handlers. When used with the EV backend AnyEvent provides outstanding performance.

Job output: ext3 filesystem. To store job output Limo simply uses the Linux ext3 filesystem, indexed by job ID and host. Since Limo is designed to not require write barriers or transactions, the dispatcher can employ native buffered file I/O, which is very fast due to the Linux kernel's integrated buffer cache and elevator algorithms. Consequently, the dispatcher does not experience significant I/O latency while recording job output, despite I/O concurrency levels in the thousands. Data redundancy is left to the OS or underlying hardware.

Metadata: SQLite, ext3 filesystem. Limo's dispatcher uses SQLite, a small relational database, to store various metadata such as users, timestamps, and exit codes. In addition, the dispatcher stores the same metadata in the filesystem using an intuitive directory structure rooted at the job ID. This ensures that nearly all the metadata can be easily recovered in the event of database corruption.

We avoided more complex relational databases because we did not need many features—Limo has only a single writer, and the overhead of maintaining a production RDBMS is considerable. SQLite, on the other hand, has adequate performance and is very easy to maintain; all data is located in a single file and can be trivially backed up or copied for experimentation.

Data interchange: JSON and YAML. JSON can provide a simple, intuitive syntax for invoking remote procedure calls and responses. C-style behavior can be attained by using a structure such as `{action: [argument, … ]}` for the request and `{code:object}` for the response. More complex response types can easily be devised and are limited only by your imagination and available memory. In other areas Limo uses YAML to store configuration, as the structure was designed to be easily editable by the user. There are many fast and well-tested JSON and YAML parsers available for nearly every programming language and environment.

Remote access protocol: SSH (secure shell). *sshd* is installed on all Yahoo! hosts and provides all the necessary features for generic task execution (authentication, encryption, terminal handling, etc). Other approaches could require users to install a custom agent or would have to be scrutinized for security. Using SSH also provides flexibility for users as it allows them to perform any task for which they are normally authorized. Moreover, *yinst*, one of our most popular tools for server maintenance, assumes it is running in an interactive shell and relies on SSH features such as port forwarding in some cases.

Architecture

The Limo architecture consists of two layers: the *dispatcher* and *workers*. Limo operates on **jobs**, which are requests to execute a command on a list of **hosts** (also called **targets**). Jobs are subdivided into **tasks**, one per host.

The dispatcher is the heart of Limo: it interacts with clients, performs necessary bookkeeping, multiplexes I/O traffic, stores job output, and issues notifications.

Workers are Limo's beasts of burden: they receive tasks from the dispatcher, log into the target hosts on the user's behalf, and relay command output. At startup, each worker makes many parallel connections to the dispatcher. Each connection is available to the dispatcher as a resource for executing a task. Provided adequate resources are available on the dispatcher, adding capacity to Limo is as simple as adding more workers.

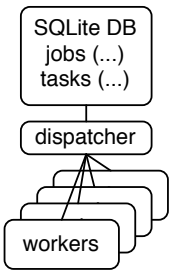

figure 1 - Limo architecture

Both workers and dispatchers are single-process, event-driven servers that handle large numbers of concurrent connections. AnyEvent handles the main event loop, executing callbacks when file descriptors become available, child processes exit, or timers expire.

Security

Because Limo operates on behalf of the user, Limo has to be entrusted with the user's security credentials. Since Yahoo! does not yet have centralized UNIX authentication the user's password must be obtained. Limo treats this information very carefully to ensure it is never disclosed improperly.

First, extensive authentication and encryption is employed by each component. The dispatcher and the workers each have separate SSL keys and certificates, and must mutually authenticate each other. In addition, the client suite ensures the dispatcher presents a valid SSL certificate each time it connects. Timestamp, source host and port are logged with each job, just as *sshd* logs them for each connection.

Moreover, the client encrypts any passwords sent to Limo using the workers' public keys. This ensures that the dispatcher cannot decrypt the password; only during worker task execution can the password be used.

Finally, passwords are never stored on disk. When necessary, passwords are exchanged via ephemeral channels such as network IPC (where they are encrypted), or UNIX pipes (where they can travel in plaintext with little risk of eavesdropping).

Pogo

Pogo retains much of Limo's dispatcher/worker architecture and underlying technologies, but expands on it in several important ways, allowing complex modeling of deployments and addressing issues of redundancy and workflow integration. Pogo is targeted towards performing tasks that are *interruptive to service* and should not be executed on too many hosts in parallel. Compared with Limo, Pogo jobs can be very long-lived, in practice often lasting for many hours or days.

Parallelism and Constraints

By design, Limo executes as many tasks in parallel as possible. Pogo, on the other hand, executes as many tasks in parallel as possible *within user-defined parameters.*

Pogo has two directives for controlling task execution - *constraints* and *sequences*. A constraint defines the maximum number of hosts in a group whose tasks may execute in parallel. A sequence defines a group of hosts whose tasks must be executed before or after another group of hosts. These directives can overlap and take on multiple dimensions, providing fine-grained control of execution order.

In order to accurately model constraints Pogo must know additional information about your hosts. On job submission Pogo asynchronously queries an external service, Culpa[1], for metadata associated with the RolesDB[2] roles your hosts belong to. Pogo requires two types of roles, *tiers* and *environments*. Tiers contain all hosts that provide a given service, regardless of environment, and share a common architecture. Environment roles contain all hosts, regardless of tier, that form a logical instance of a deployment. Each tier role must be given a name and each environment role must be given a type and an id. Pogo uses constraint, sequence, tier and environment information to build a matrix of host availability and compute a sequence of tasks.

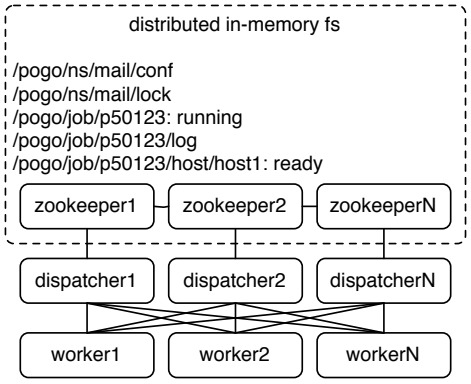

figure 2 - Pogo architecture

Redundancy with ZooKeeper

Pogo dispatchers are redundant: any dispatcher can fail or be restarted without interrupting job or task execution. This is achieved by storing all state, including job state, host state, configured and computed constraints and sequences in ZooKeeper. ZooKeeper is an open-source Yahoo!-sponsored Apache Foundation project that provides a virtual distributed in-memory filesystem and a small number of synchronization primitives. Pogo

configuration is stored in ZooKeeper and constraints are defined on a per-namespace basis (figure 2, 'ns'). The dispatchers maintain locks on hosts in ZooKeeper when executing tasks, limiting the number run in parallel.

Task Logic

Pogo contains an intelligent "stub" program that workers copy to and run on each target. The stub implements pre- and post-task logic, called *hooks*. When configured to do so in the job options, the stub will execute scripts in `/home/y/etc/pre.d` sequentially before running the command. If all pre-hooks and the command succeeded by returning `0`, the stub will execute scripts in `/home/y/etc/post.d`. All hooks as well as the command must succeed or the task will be marked as failed.

The stub also implements retry logic and detailed output parsing such as fine-grained timestamps and differentiation of standard output and standard error streams.

Individual tasks can be retried (in the case of transient failures) or skipped (in the case of known issues) allowing the operator some degree of freedom to override constraints based on real-world conditions.

Workflow Integration

Pogo allows modeling of commands in *recipes* and sharing of recipes in *cookbooks*. Recipes are written in YAML and describe some or all the options associated with a job. Cookbooks are YAML files containing one or more recipe. Pogo can remotely load cookbooks via HTTP allowing large teams to easily share recipes for code review or debugging. CMRs can reference specific recipes rather than trying to describe complex procedures in a web form.

Pogo-UI adds web-based job visualization based on YUI and canvas. Users can start from a high-level view of tens of thousands of hosts and drill-down to individual hosts and watch job output similar to `tail -f`.

Finally, an external service, VHL, adds a Vespa-indexed, VDS-based data store for job and task information, including searchable full-text task output. On job completion a callback is executed that spools all job metadata and host output for indexing. Once indexing is successful, jobs are purged from the live system and replaced with pointers to the VHL documents.

Example 1: A Typical Limo Job Submission

SE Homer Simpson wants to run `yinst restore -igor` on every power.yahoo.com front-end webserver. He wants to do this quickly so he can call it a day and go to his favorite watering hole, Moe's. Homer installs the limo_client package and runs `limo-run –I @power.front-end yinst restore –igor`. *limo-run* collects user and package passphrases, then queries RolesDB to expand the `@power.front-end` range into a list of hostnames and submits a JSON-encoded message containing the run command and the job parameters to the dispatcher.

After the dispatcher validates the request it assigns a unique job id and stores the relevant job data. then returns the id back to Homer's client. Homer can then use the id to check the status of the job and obtain its output.

The dispatcher then subdivides the job into multiple tasks, one per target host, and finds a free worker connection capable of executing each task. Workers get instructions to execute the tasks as JSON-encoded objects. The worker then forks and executes the command on the target host via `ssh`. standard output and standard error streams are captured and forwarded back to the dispatcher where they are recorded to the filesystem.

As each *ssh* session finishes, its exit code is returned by the worker to the dispatcher, which records it and marks the task as completed. Once all tasks have finished the dispatcher notifies Homer via Yahoo! Messenger and/or email. Homer can then check the status of his job via the *limo-status* command, or print or save its output via the *limo-cat* command.

Example 2: Single Dimension Constraints

Homer has 20 HTTP servers divided into two clusters of 10 hosts each. Members are defined in RolesDB and additional metadata is kept Culpa. For simplicity's sake, all tasks will take exactly the same amount of time to complete, and all will complete successfully.

`@power.front-end` contains all hosts that provide http service, and `@power.env.ac4` describes a logical serving unit of `@power.front-end`, perhaps all hosts behind a single load balancer (figure 3)

Since Homer is a simple button-pusher, he has no detailed knowledge of the details of the servers he operates. But the engineers at his plant know that in order to avoid interrupting service, only 20% of each cluster can be removed from service at any time, so they declare a constraint in Pogo (figure 4)

Homer then runs `pogo run -I @power.front-end yinst restore -igor`. The dispatcher will look up metadata for all the roles of each host. Finding a constraint on the "cluster" environment, each roles is expanded and "20%" is resolved to an absolute number, 2. Tasks are issued for the first two hosts in each cluster (figure 5, run group A). As soon as any task completes successfully the dispatcher will issues a new task for the next host in the sequence. If a task fails, no subsequent task will be issued unless the user issues a retry or skip command for the failed host. This ensures that minimally 80% of hosts will be in service in each cluster.

```
@power.front-end:
  members: http[1-10].power.ac4.yahoo.com
           http[1-10].power.sp1.yahoo.com
  role-type: tier
  tier-name: web

@power.env.ac4:
  members: http[1-10].power.ac4.yahoo.com
  role-type: env
  env-type: cluster
  env-id: ac4

@power.env.sp1:
  # configured similarly to @power.env.ac4
```

figure 3 - role metadata

```
constraints:
  web:            # tier-name
    cluster: 20%  # environment-type
```

figure 4 - constraints

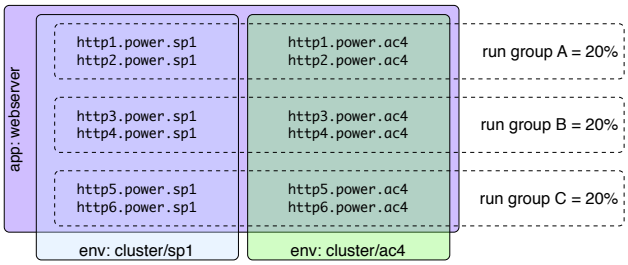

figure 5 - run groups

Example 3: Sequences

Another tier is added, 'db' (figure 6). These represent back-end servers that must be upgraded before any front-end servers in the environment. To ensure this, we write a sequence constraint (figure 7).

```
@power.all
  members: @power.back-end,
           @power.front-end
@power.front-end:
  members: http[1-10].power.ac4.yahoo.com
           http[1-10].power.sp1.yahoo.com
  role-type: tier
  tier-name: web
@power.back-end
  members: db[1,2].power.ac4.yahoo.com,
           db[1,2].power.sp1.yahoo.com
  role-type: tier
  tier-name: db
@power.env.ac4:
  members: http[1-10].power.ac4.yahoo.com
           db[1,2].power.ac4.yahoo.com
  ...
```

figure 6 - role metadata

```
constraints:
    web:
      cluster: 20%
    db:
      cluster: 1

  sequences:
    cluster:
      - [ db, webserver ]
```

figure 7 - sequence constraint

Now sequences will be enforced (figure 8). Run groups A and B ensure "db" tasks are executed first and one at a time. Run groups C,D proceed with the "web" constraints.

The same ordering happens in both sp1 and ac4 clusters, ensuring maximum parallelization without service interruption.

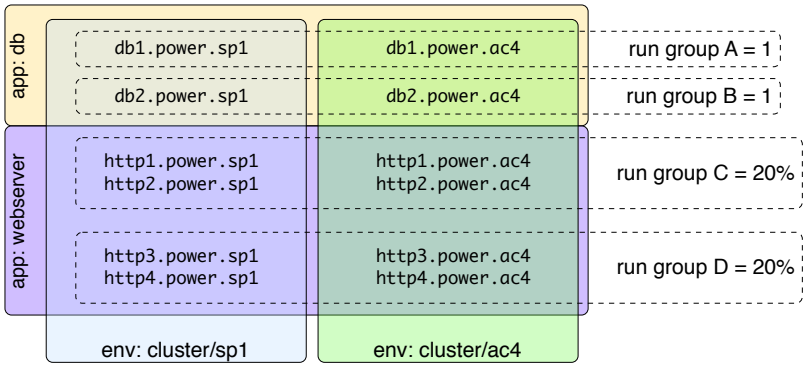

figure 8 - run groups

Adoption

Since its original release in 2007, Limo has gained increasing popularity at Yahoo!. Awareness of its existence and usefulness spread primarily by word of mouth from the MP&S SE team (for whom it was originally written) to the SE teams for Metro, Coke, Mail and beyond. Today, Mail is by far the largest user of Limo, and has added substantial functionality to it in the form of Pogo.

For the 1-month period ending June 13th (data from vhl.corp.yahoo.com):

Limo jobs issued: 7353

Limo tasks executed: 515779

Properties using Limo: 80+

Pogo jobs issued: 3084

Pogo tasks issued: 300282

For comparison, Taxi tasks for the same period: 41816

Conclusion

Originally conceived as a high-performance, general-purpose parallel shell execution engine, Limo has gained substantial functionality via the Pogo constraint system. Pogo also brings together many other Yahoo! technologies to create a robust, coherent and operable system.

Despite massive growth in utilization, Limo's simple design has stood the test of time, requiring very little hardware to provide a great deal of power to service engineers at Yahoo!. We hope both tools generality, openness and robustness will allow imaginative tool authors to expand their power even further. We hope the principles that drove its design will influence software engineers throughout the company.