

---

# Beanstalks: A General Purpose Job Queue for the Computer Vision Cluster

---

Yangqing Jia, Hyun Oh Song  
Department of EECS, UC Berkeley  
{jiayq, song}@eecs.berkeley.edu

## Abstract

The computer vision group at Berkeley EECS has a set of computers for research use, yet lacks an efficient framework to utilize the computational power. In this project, we propose a novel job queue system that is designed for the general-purpose use in the group. Specific needs of the group such as being easy to use, multiple language support, and tolerance of different user styles are considered and reflected in the system design. We also discuss typical crash recovery and load balancing issues, and proposed ways to implement them. Finally, we discuss the experience and lessons learned during the CVPR paper deadline, and propose several ways to further improve the current system.

## 1 Introduction

The computer vision group at Berkeley EECS and ICSI, lead by Prof. Trevor Darrell, has a set of computers for research use, which the members usually refer to as the “vision cluster”. These clusters provide the daily computational power for about 20 students and postdoctoral fellows. However, although the collection of these machines is called a “cluster”, the machines are not organized as a real cluster that can easily carry out distributed computing. Usually, a user logs on to one of the machines, executes codes on the machine, and manually checks if the machine has finished to task or not. Such operation is highly inefficient and problematic, so we would like to have a general-purpose system that frees us from manually distributing the jobs and monitoring the progress. This inspires us to design Beanstalks, a job queue system running on the vision cluster.

The purpose of designing Beanstalks is two-fold. In the first place, we aim to obtain a hands-on experience of a simple system design as computer theory students, which hopefully fulfills the CS262a course requirement. The grand goal, however, is to have Beanstalks actually running on the cluster as a service that everyone will use in the daily work. By the time this report is written, we believe that Beanstalks is already able to support simple uses such as parameter searching, and is empirically proved effective and timesaving. We designed the system during the semester and actually applied it during the month before the CVPR paper deadline, receiving positive feedbacks from the users.

In this report, we will discuss the background and implementation details of Beanstalks, the experience and lessons learned during the time, and possible future directions. The report is organized as follows: Section 2 discusses the problem we aim to solve in detail; Section 3 explains the detailed design of Beanstalks. Section 5 provides toy-case experimental results showing the effectiveness of the system, and discusses the experience and lesson learned during the CVPR paper deadline. Section 6 concludes the paper.

## 2 Problem Statement

The vision cluster lies in a protected gigabit LAN under the ICSI infrastructure. One machine called the `exp-door`, maintained by ICSI, serves as the gateway to the local LAN. As of October 2010, the cluster contains 3 bookkeeping computers, 2 high-cpu high-memory computers, one GPU computing machine and 24 normal workstations. All these machines are equipped with Ubuntu Linux. A few features of the cluster are:

1. All the machines are multi-core machines, with 4 or 8 cores each and 4-8GB memory.
2. Other than `exp-door` which is maintained by ICSI staff, crash of individual machines is common - though not very frequent - as the cluster is maintained by whoever is in the group and happens to have time fixing them.
3. A centralized storage is provided using NFS, and are mounted on all the machines using the same path. The storage is provided using commodity RAID5 boxes, with daily backup to ensure data safety.
4. Most research codes running on the cluster are coded single-core, i.e., a  $n$ -core machine can run  $n$  pieces of our research codes.
5. Most of the research work can be done without a graphical user-interface. In fact, most people use commandline ssh only.

A typical task often emerging from our daily work is to execute several similar tasks, with slightly different parameter settings. Often, we want to do a parameter search to find the optimal parameter for a specific object classification/detection task: the same classifier is used, and performance is tested by running this classifier with different parameters. Each run is independent from other runs, and running them simultaneously on different computers (or CPU cores) will save us much time.

Before Beanstalks is deployed, people need to log on to each machine manually, and execute commands /monitor the progress of the execution manually. This is particularly inefficient and problematic:

1. People tend to choose machines that are more “powerful”. As a consequence, the high-cpu high-memory machines are constantly swamped while the normal workstations are never used to its full capacity.
2. There is usually a long time window between the finish of the previous execution and the next execution when carried out manually, leading to inefficiency.
3. People need to manually check others’ activities to prevent conflict.

These are the main reasons we implemented Beanstalks. We design the system in such a way that it meets (or we hope to make it meet) the following requirements:

1. It is easy to use: the system takes minimal learning time and minimal modification to the current codes. This would allow people to still focus on their research instead of spending significant time adjusting to the new system.
2. It supports multiple programming languages: a large variety of languages including C/C++, Python, Matlab, and Java are used in our group, and some codes are hard to port to other languages, so this is a must.
3. It should not harass users that do not use this system. There are members in the group that still wants to use the old way, and we often want to “try out” some codes, for which using Beanstalks will be an overkill. Thus, the existence of Beanstalks and the old manual way should not conflict.

### 2.1 Related Works

Parallel programming has been a prevailing trend these years, including a wide range of frameworks from shared memory programming such as OpenMP [4] to distributed computing using message passing, to the recent idea of cloud computing [2]. Some of these frameworks involve deep language-level features, while some of these frameworks are more higher-level. A popular programming

model that works with multiple computers in a cluster is the MapReduce [5] model: it decomposes the whole task using the Map() operation to independent subtasks that can be carried out in parallel, and collects the results using the Reduce() operation. This is most similar to what Beanstalks is trying to achieve: in our work, we aim to distribute the subtasks of a parameter tuning task to different machines, and to achieve this using a relatively simple user interface.

However, we would point out that Beanstalks is not a programming language: it works on the application level instead of the programming level. It does not attempt to require the user to re-write the program, but works by wrapping the execution of programs as a “job” which is represented by a shell string. The reason why we abandoned all the parallel programming frameworks is that all of them require the user to immerse him/herself in learning the programming details before s/he can do anything useful. This is prohibitive in our case, as people would tend to stick to their old way, especially when the group is focused on research instead of programming.

On the other hand, Beanstalks is closely related to several high-level models and other techniques. We designed our model in a highly modular way similar to the Click Router [6]. In fact, all the modules in Beanstalks can find its similar counterpart in Click. As we will discuss later, lottery scheduling [7] can be adopted to perform resource management in Beanstalks, when enforcing user quota is necessary.

### 3 The Modules

#### 3.1 Overall View

In general, Beanstalks is a job queue system that is composed of three main modules: the client, the queue/monitor, and the workers. The queue/monitor module maintains the job pool, which is essentially a job queue with priority, and monitors the health of individual workers. Users use the client module to submit jobs to the queue, and the workers retrieve jobs from the queue and execute them. Figure 3.1 shows the block diagram of the proposed framework. Notably, Instead of having the queue pushing jobs to individual workers, we have the workers pulling jobs from the queue. In another word, the distribution of jobs is handled by the workers instead of the queue. We will explain the justification in the later parts of the section.

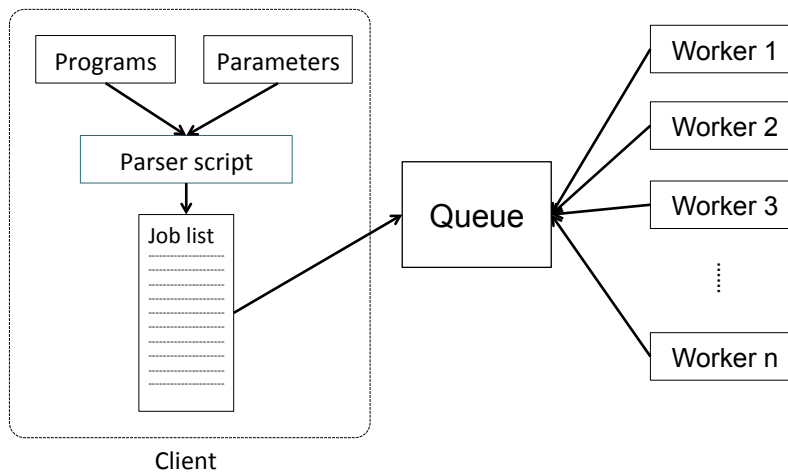


Figure 1: The block diagram of the proposed framework

#### 3.2 The Queue Module

The fundamental component of Beanstalks is a priority queue. We employed the open-source beanstalkd program [1] as the base queue. The queue implementation provides several features: it naturally supports priority which is represented as an integer number from 1 to  $2^31$ , and has built-in crash recovery to recover the content of the queue after a possible system crash. Each element in

the queue is a string. There is no restriction imposed on the string other than the length (a maximum of 65536 bytes is supported).

We allow multiple job queues to exist simultaneously. Each job queue may correspond to one task which is a batch of jobs, so there is a correspondence between the job queue and the actual task (or user). A job queue is composed of two subqueues: one called the submission-queue to which the client submits jobs, and the other called the return-queue to which the workers return result messages.

In practice, the queue is run on the `exp-door` machine. The reason is that as an ICSI-maintained machine, `exp-door` is the machine in our cluster that has the smallest downtime, and crashes most infrequently. We thus make a reasonable assumption that the queue is never down - such assumptions are also made in other systems such as MapReduce, so we believe that it is reasonable<sup>1</sup>. Also, since `exp-door` is a Pentium III machine, any heavy computation will significantly decrease the speed, and this is why we choose to leave most operations, like load balancing, to the individual workers who has better computation resources.

Without loss of generality, we will call the machine as “the queue” if it does not introduce confusion.

### 3.3 The Protocol

To ensure portability, all the jobs and interactions between machines including the client, the queue and the workers are formatted as a string that is passed along different subqueues in the queue module. We will call each string as a message. Each message is composed of a message header, which is a three-letter keyword, and an arbitrary substring that carries whatever information that needs to be passed on. For example, a job message that defines a dummy job to be executed by one of the workers would look like

```
RUN sleep 10
```

where the “RUN” keyword defines that it is a job to be executed.

Any program, as long as it can be called by a shell command, can be wrapped as a job. For example, a matlab program may be wrapped as `RUN matlab -r m-file-to-be-run`, and a java program can be wrapped as `RUN java program-to-be-run`. Compiled C programs and python programs can be wrapped even more simply. This enables us to allow existing programs to be naturally executed under our framework, and we believe that the flexibility is already enough for our daily use.

The worker that carries out this job, if success (in the above case it’s most likely to be a success) will return a result message that looks like

```
SUC melon4-23102 run sleep 10 logged at
    /vision/beanstalks-log/melon4-23012-2010-11-11-11:11.11.log
```

which is composed of the “SUC” header, followed by the worker name, the job that is carried out, and where the log of the execution is. We will put the detailed explanation of different parts of the protocol in their respective modules’ subsections.

### 3.4 The Workers

A worker is a daemon program that periodically checks all the job queues and retrieves jobs if there is one. Specifically, when there is a job, a worker dequeues that job message, and carries out the job. If it’s a success, it returns a successful message to the corresponding return-queue as:

```
SUC [worker-name] run [command-executed] logged at [logfile-path]
```

If it’s a failure (mostly because the command is incorrectly written, or the program being executed has bugs), it returns a failure message by replacing the `SUC` header with the `FAL` header. In both cases, the worker will delete the job message from the queue. The reason why we don’t return a

---

<sup>1</sup>In fact, if `exp-door` is down, there is no other way (even manually) to log on to the cluster.

failed job back to the queue (which might be the most direct thought) is that the failure is usually not due to Beanstalks: it is most likely that re-executing the job will still get a failure, so it is better off to not execute it again and inform the user about the problem. The user may further fix the problem and submit new jobs later.

As a simple load balancing method, for each machine with  $n$  cores, we launch  $n - 1$  workers, so that these workers occupy at most  $n - 1$  CPU cores at the same time<sup>2</sup> (Note that we assume that most programs run on a single core only). Further load balancing designs will be discussed in the next section.

### 3.5 The Client

The client module is the only module that the user will use. Essentially, what it does is to take in a text file, where each line is a shell command corresponding to a job. The client module reads the file, wraps the jobs with the priority the user specifies, and submits them to the queue. For easy to use purpose, it is written as a python script, and any computer that has python installed and is able to access Internet can submit jobs to the queue, without the hassle of logging on to exp-door, create queue operation sessions, etc. Of course, the user still needs a valid vision cluster account to use the system.

As one of the main uses of Beanstalks is parameter tuning, we also designed a simple parameter parser that takes in a simple command template with parameter positions and a list of parameters to be tried out, and outputs the jobs to be submitted. For example, one may perform a grid search over two SVM parameters  $C$  and  $\gamma$  by the following input:

```
svm -train -kernel rbf -C [1] -gamma [2]
[1] 0.001, 0.01, 0.1, 1, 10, 100
[2] 0 0.25 0.5 0.75 1
```

The user is recommended to use a task-related queue name for each batch of jobs. This enables us to figure out what is being carried out in the system. After submission, the client then monitors the return messages from the worker. As for now, the returned messages are simply status reports to indicate whether all the jobs has been carried out or not, and we will explore the possibility of returning richer information in the future.

## 4 Design for Practicality

In this section we discuss two aspects to keep Beanstalks practical for daily use: crash recovery and load balancing.

### 4.1 Crash Recovery

The Beanstalkd queue has a simple crash recovery mechanism built in. For each job, the client can specify a “time-to-run” (TTR) value that estimates the time that is needed to run the job. The time starts to count when the worker reserves the job, and when the time is up and the worker has still not finished the job, the job is returned to the queue to be re-executed.

Such a mechanism does provide a bottomline support for recovery when a worker crashes. However, in practice we believe that it is not reasonable, for the following two reasons:

1. It is difficult for the user to estimate the TTR of a program. Usually, a SVM classifier training which is common in our daily tasks takes sometimes as short as one hour and sometimes as long as a day, and it is difficult to pre-estimate before actually trying out the program. A TTR estimated too short will make a program being re-executed again and again, wasting the computation resource.
2. The previous problem can be addressed by setting a TTR large enough (e.g. two days). However, this renders the crash recovery mechanism virtually useless: it does make sense

---

<sup>2</sup>We always leave out one CPU core so that we will still be able to log on and do maintenance work.

to have the worker crashing and the corresponding job returned to the queue only two days later. When it's approaching a paper deadline this would be unacceptable.

Thus, we decided to use a different crash recovery mechanism. To this end, we specify a queue called `newcomers`. When a new worker is launched, it first sends a newcomer message to this queue:

```
NEW [worker-name]
```

to declare its existence. When a worker exits normally, it sends a goodbye message to the queue<sup>3</sup>:

```
LLP [worker-name]
```

A lightweight monitor program running on `exp-door` monitors this queue, and maintains a list of current workers. It then creates one queue for each worker named after the worker name to interact with each worker. Periodically (in practice we set this to be 10 minutes), it sends a message containing only the header `RPT` to request the users to report its status. The worker then reports back as follows:

```
STS [worker-name] job [job-body] priority [priority-value]
```

Also, when a worker moves on to a new job, it sends a `STS` message to the monitor to get it updated. Once the monitor detects that a worker is non-responsive (i.e., the machine that worker is on crashed), it deletes the worker from its bookkeeping list and returns the last job the worker is working on back to the queue. This enables us to ensure every job to be carried out, as well as to avoid unnecessary duplicate runs.

## 4.2 Load Balancing

It is important to keep the load balanced between the workers. Before the CVPR deadline (November 2010), the load balancing system is solely based on controlling the number of workers on each machine. This is proved to be enough when all the users are only using Beanstalks to run jobs on the cluster, but soon we found several issues, two most important ones are as follows:

1. Users that do not use the job queue system complains that Beanstalks is occupying all the computers, leaving no resource for them to run their programs. Of course, one can ignore the CPU load and force the program to run, but with more programs than CPU cores running on one machine, one is sacrificing overall efficiency and is risking crash of the machine. Actually, the remote login using `ssh` will become as slow as being non-responsive, and since programs run as long as a day, it is hard to estimate when to resume the interactive `ssh` session.
2. When we switched to a newer version of Matlab, problems start to emerge: even when there is no multi-core programming in our Matlab code, the built-in functions such as eigenvalue computation is inherently multi-core. Moreover, Matlab removes from the preference panel the option to choose how many cores it will use, leaving us no way to restrict the use of cores.

The two reasons above forced us to implement an additional load balancing scheme that looks into the actual CPU core usage and determine if it is reasonable to launch a new job. Specifically, for each worker, when it is ready to get a new job, it first checks the CPU usage: if there are still some available CPU cores on the machine. If there is one, the worker then proceeds to retrieve a job. Otherwise, it waits for a while and then rechecks the CPU usage. Also, when it's idle, it reports to the monitor that no job needs to be returned to the queue should it crash during the time. We tested this load balancing scheme against the old method in the experiments section.

## 5 Experiments and Discussions

In this section, we quantitatively check the throughput of the system and the performance of load balance, and discuss the qualitative lessons learned from the CVPR paper deadline.

---

<sup>3</sup>We want the worker to be a little Vulcan, hoping that it would be more *logical*.

## 5.1 Experiment 1: Throughput

We used dummy jobs that ask the worker to do nothing to test the throughput of the system, i.e., with all the bookkeeping mechanisms, how fast jobs can be passed via our system. To this end, we used one client, and varied the number of workers to evaluate the throughput under different cases. Every time we submit 8,000 dummy jobs and plot the elapsed time vs. the number of jobs being processed in Figure 2.

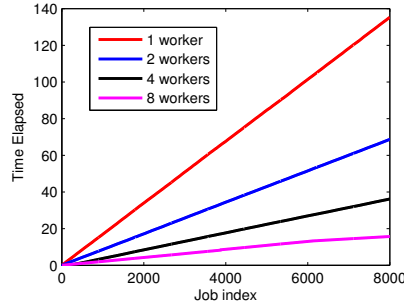


Figure 2: Elapsed time vs. the number of jobs being processed, with 1, 2, 4, and 8 workers respectively.

It can be observed that one worker is able to process approximately 65 jobs per second. Given the fact that each job takes hours to run in our daily work, the overhead is negligible. Note that the capacity of the queue is much larger than 65 jobs per second: when we use 8 workers, the processing time is about 8 times faster than using 1 worker, indicating that the lightweight queue is actually able to support a much larger throughput. In practice, we launched about 180 regular workers on our cluster, and did not find any noticeable lag that is caused by the overhead of the system.

## 5.2 Experiment 2: Load Balancing

We tested the load balance technique as described in the previous section. To this end, we launched four workers on three 4-core machines `apricot2`, `apricot3` and `apricot4`. `apricot3` was running two external jobs that occupied two of the cores, simulating the case when an external user is logged on to the computer and is not using `Beanstalks`. Then, we tested the number of jobs completed by the machines, using a dummy job that requires the worker to count from 1 to 10,000.

Figure 3 shows the number of jobs being executed per worker and per machine. Without load balancing, all the machines executed approximately the same number of workers, with `apricot3` slightly less since its operation is slower due to the two external jobs running. Still, 6 jobs being carried out simultaneously on the 4-core `apricot3` significantly affected the speed. Ideally, we would want `apricot3` to carry out fewer jobs - about half compared against `apricot2/4`. This is achieved by the load balancing mechanism as shown in the figure.

## 5.3 The CVPR Experience, and Future Work

During the CVPR deadline, we deployed our system for experiments of the paper submissions in our group. The result turned out to be successful: overall we had more than 2,000 jobs carried out, and the machines with `Beanstalks` workers running were able to keep a total CPU usage of over 75%. We did find several issues, such as load balancing as described and fixed above. Some experiences led us to consider several possible future directions, which we describe below.

**Job Cancellation.** In the current system, when the client submits the jobs, it is done with them. Even if the client accidentally submitted some wrong jobs, all it can do is to wait until the jobs are executed by the workers, possibly with `FAL` returns. An easy-to-use interface to cancel submitted (and maybe running) jobs is much desired. Actually, it is not too hard to implement - for the jobs already in the queue, a scan through all the jobs can filter out those jobs needed to be deleted and

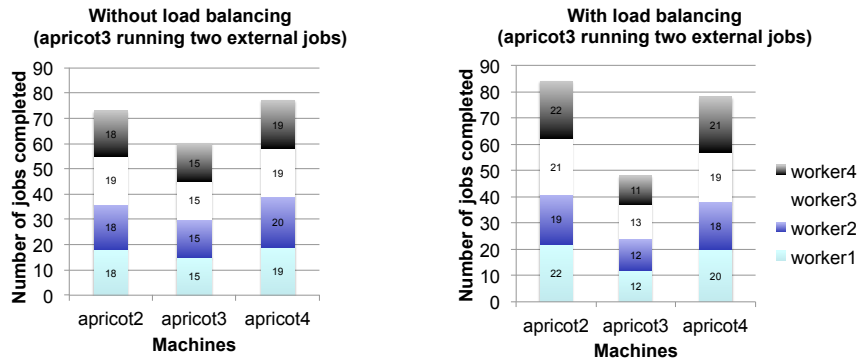


Figure 3: The number of jobs being executed per worker and per machine. Left without load balancing and right with load balancing.

return the good jobs back to the queue; for the jobs already running, the client can notify the monitor who further notifies the corresponding workers to abort the execution.

**Worker reservation.** Sometimes, a specific task needs to run programs on a subset of the machines only. For example, some GPU computation program can only be carried out on computers with GPUs. In this case, we would like the client to reserve the computers first, and then carry out computation. This can be achieved by creating job queues that are not checked by regular workers, and reservation simply means to tell some workers to check these job queues.

**User Quota.** A practical problem is that some users may submit much more jobs than others, thus blocking the other users to use the system. A user quota would help fairly distribute the computation power to all the users. One way to do this is to have different queues for different workers, and assign weights to the queues: a queue with a higher weight get checked out more often. The weight calculation can be essentially based on a fair-share scheduling mechanics. Further, lottery scheduling [7] can be adopted to dynamically compute the weight.

**Experimental Design.** The parameter tuning in our daily work involves the classical problem of experimental design [3]. Currently, we are doing a simple grid search to find the optimal parameters. Of course a more efficient way is to automatically adjust the parameters to be tested based on the feedback of the currently finished jobs. To make Beanstalks a more intelligent experimental design system, we may need to (1) enrich the information returned by the workers; (2) enabling more complex job submission and cancellation; (3) have a more intelligent way to select parameters based on application-based knowledge. The first two can be achieved by improving the current system, and the third can be implemented as a client add-on similar to the parameter parser we are currently using.

## 6 Conclusion

In this project we proposed Beanstalks, a novel job queue system that is designed for the general-purpose use of the computer vision group at Berkley EECS. we discussed specific needs of the group and the way we designed the system to meet these needs. The system is put to use during the CVPR deadline, and is a regularly running service on the cluster when the report is written. We discussed the experience and lessons learned, and proposed several ways to further improve the system.

## Acknowledgement

We would thank Sergey Karayev, who is also from the vision group, for his extensive participation in the design and implementation of Beanstalks. We would also like to thank Prof Eric Brewer for valuable recommendations of how to improve the system in the future. The Beanstalkd queuing

daemon that we used is an excellent open-source project, originally designed for distributed web services.

## References

- [1] <http://kr.github.com/beanstalkd/>.
- [2] M. Armbrust, A. Fox, R. Griffith, A.D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.
- [3] G.E.P. Box, J.S. Hunter, and W.G. Hunter. Statistics for experimenters: design, innovation, and discovery. 2005.
- [4] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 2002.
- [5] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
- [6] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M.F. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems (TOCS)*, 18(3):263–297, 2000.
- [7] C.A. Waldspurger and W.E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, page 1. USENIX Association, 1994.