Spring 2006 CS739 Project Write-up: A New Architecture for Matchmaking in Condor®

Todd Tannenbaum
tannenba@cs.wisc.edu

Department of Computer Sciences
University of Wisconsin-Madison

May 12, 2006

Abstract

We have designed and implemented a new architecture for matchmaking within the Condor High Throughput Computing system. This new architecture preserves all of the policy semantics of Condor, but has superior performance characteristics with large numbers of users and submitted jobs. Furthermore, our architecture enables the matchmaker to view a global state of the system (all machines as well as all jobs) in order to make better matches. Finally, we made modifications to Condor in order to accurately model and measure the matchmaking behavior of a large installation through the use of trace files, and used this mechanism to test the validity of our architecture changes.

1 Introduction

Condor is a distributed job and resource management system [3]. Compute jobs and machines are represented by ClassAds, and a centralized matchmaking service matches jobs to machines in accordance with provided policies. Unfortunately, the centralized matchmaking service has become a significant performance bottleneck in many Condor installations.

The *Condor High Throughput Computing* system is a distributed job and resource management system. Users submit jobs into the system, and Condor matches the job with an appropriate machine for execution.

In the Condor system, jobs and machines are represented by *ClassAds*, and submission hosts are connected to execution hosts via a ClassAd matchmaking service. While many other services in Condor are designed as distributed components, the matchmaking service is centralized. Furthermore, it has become a significant bottleneck in many Condor installations, especially if the job mix includes large numbers of submitted jobs and/or bursts of jobs. A Google search for ”Condor long negotiation cycle site:lists.cs.wisc.edu” reveals that the matchmaking performance bottleneck is not only a theoretical limitation in Condor, but is causing real-world angst.

The next section gives some very brief background information about Condor and ClassAds, although the reader is encouraged to refer to other cited publications for the salient details. Section 3 on page 3 presents our projects goals and requirements, section 4 on page 4 presents our design, and section 6 on page 7 explains how we evaluated our work.

2 Background

2.1 ClassAds

A ClassAd is a set of uniquely named expressions, using a semi-structured data model so that no specific schema is required by the matchmaker. Each named expression is called an *attribute*. Each attribute has an *name* and an *value*. The first version of the ClassAd language allowed simple values such as integers and strings, as well as expressions comprised of arithmetic and logical operators.

Job ClassAd

```
[
MyType = "Job"
TargetType = "Machine"
Requirements =
((other.Arch=="INTEL" &&
other.OpSys=="LINUX")
&& other.Disk > my.DiskUsage)
Rank = (Memory * 10000) + KFlops
Cmd = "/home/tannenba/bin/sim-
exe"
Department = "CompSci"
Owner = "tannenba"
DiskUsage = 6000
]
```

Machine ClassAd

```
[
MyType = "Machine"
TargetType = "Job"
Machine = "nostos.cs.wisc.edu"
Requirements =
(LoadAvg <= 0.300000) &&
(KeyboardIdle > (15 * 60))
Rank =
other.Department==self.Department
Arch = "INTEL"
OpSys = "LINUX"
Disk = 3076076
]
```

Figure 1: **Two Sample ClassAds from Condor.**

After gaining experience with ClassAds, we created a second version of the language that permitted richer values and operators permitting complex data structures such as records, lists, and sets.

Because ClassAds are schema-free, participants in the system may attempt to refer to attributes that do not exist. For example, an job may prefer machines with the attribute `(Owner == ''Fred'')`, yet some machines may fail to define the attribute `Owner`. To solve this, ClassAds use *three-valued logic* which allows expressions to evaluated to either `true`, `false`, or `undefined`. This explicit support for missing information allows users to build robust requirements even without a fixed schema.

The Condor matchmaker assigns significance to two special attributes: `Requirements` and `Rank`. `Requirements` indicates a constraint and `Rank` measures the desirability of a match. The matchmaking algorithm requires that for two ClassAds to match, both of their corresponding `Requirements` must evaluate to `true`. The `Rank` attribute should evaluate to an arbitrary floating point number. `Rank` is used to choose among compatible matches: Among provider ads matching a given customer ad, the matchmaker chooses the one with the highest Rank value (noninteger values are treated as zero), breaking ties according to the provider's Rank value.

ClassAds for a job and a machine are shown in Figure 1. The `Requirements` state that the job must be matched with an Intel Linux machine which has enough free disk space (more than 6 megabytes). Out of any machines which meet these requirements, the job prefers a machine with lots of memory, followed by good floating point performance. Meanwhile, the machine ad `Requirements` states that this machine is not willing to match with any job unless its load average is low and the keyboard has been idle for more than 15 minutes. In other words, it is only willing to run jobs when it would otherwise sit idle. When it is willing to run a job, the `Rank` expression states it prefers to run jobs submitted by users from its own department.

For further information about ClassAds and the ClassAd language, see [6, 5].

2.2 Existing Matchmaking Architecture in Condor

Within a given Condor installation, one machine will serve as the pool's central manager. In addition to the `condor_master` daemon which runs on every machine in a Condor pool, the central manager runs the `condor_collector` and the `condor_negotiator` daemons. Any machine in the installation that should be capable of running jobs should run the `condor_startd`, and any machine which should maintain a job queue and therefore allow users on that machine to submit jobs should run a `condor_schedd`.

Condor allows any machine to simultaneously execute jobs and serve as a submission point by running both a startd and a schedd. Figure 2 displays a Condor pool where every machine in the pool can both submit and run jobs, including the central manager.

The interface for adding a job to the Condor System is `condor_submit`, which reads a job description file, creates a job ClassAd, and gives that ClassAd to the schedd which manages the local job queue. When the schedd realizes that it needs another machine, it triggers a *negotiation cycle*. During a negotiation cycle, the negotiator queries the collector to discover all machines that are willing to perform work and all users with idle jobs. The negotiator communicates *in user priority order* with each schedd that has idle jobs in its queue, and performs matchmaking to match jobs with machines such that both job and machine ClassAd requirements are satisfied and preferences (rank) are honored. During this process, the schedd sends one job ad at a time to the negotiator, which responds with either a matching startd ad or with a no-match-found response. The schedd responds, in turn, by sending its next highest priority job. The negotiation cycle continues until all machines have been allocated, or

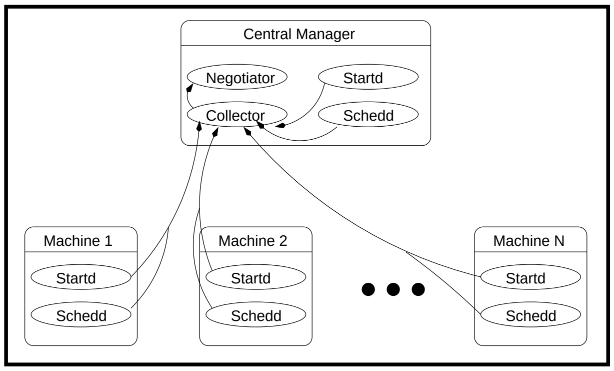

Figure 2: **Condor Architecture.**

until the schedds have sent each job in their queue to the matchmaker for consideration.

An important point to note is that the negotiator has only one job at a time in its memory, and only considers one job at a time.

Further explanation of Condor's existing architecture and matchmaking framework has been published in [7, 4, 8].

3 Goals and Requirements

We had several goals in mind for this project. First and foremost was to improve performance by reducing the time required to perform a negotiation cycle. However, we felt it was very important to *preserve all the semantic guarantees* made by the original matchmaking architecture. This requirement prevented us from exploring techniques such as limits upon the amount of time the negotiator could communicate with a given schedd, or similarly to limit the number of jobs a schedd would send to the negotiator. While such simplistic techniques could be effective, they would not ensure the same policy semantics which have helped to make Condor such a popular system with site administrators. For instance, a limit on the number of jobs a schedd could send to the negotiator would break the proper operation of the machine ClassAd's `Rank` expression.

A second goal of our project was enable the negotiator to make better globally optimized decisions by having efficient access to all of the job ClassAds at once. In the original system, the negotiator has all of the machine ClassAds, but only considers one job ClassAd at time. This prevents Condor from considering an entire families of possible scheduling algorithms, such as stable marriage [2]. It also limits the quality of the matches with respect to system-wide throughput. As an example, consider a pool of 10 machines where only one machine has more than 512 MB RAM. If user A submits a job with no requirements, and user B submits a job that requires more than 512 MB of memory, Condor may arbitrarily give the large memory machine away to user A because the negotiator had not yet "seen" the job request from user B. In the current system this situation currently results in either wait-while-idle or an eventual expensive job preemption (where user A's job is killed and replaced with user B's).

Finally, we desired a solution that would preserve the lightweight footprint of the current negotiator design. Specifically, we wanted to keep the negotiator simple, stateless, and with a low resource consumption. Keeping the negotiator as a simple, stand-alone single process allows it to be deployed on-the-fly as needed at remote locations via a technique called GlideIn [1]. A negotiator that is heavyweight, such one that required a full-fledged RDBMS installation or a complicated installation process, would inhibit GlideIn techniques. Keeping the negotiator stateless is a major advantage for availability; currently if the machine running the negotiator fails, Condor simply re-elects a new machine and restarts the negotiator. This is a relatively simple process because of the negotiator's stateless nature. Resource consumption also should be kept low even in the event of many thousands of submitted jobs.

4 Design

In the design phase of our project, we explored several techniques with varying degrees of success. Our design was guided by our own past experience and familiarity with the current architecture, as well as by collecting and analyzing profile data of the current negotiation cycle (details on what was data collected, see section 6 on page 7. We ultimately settled upon the below approaches, each of which is discussed in its own section:

- Autoclustering
- Negotiator MatchList Caching
- Advertisement of Resource Requests

4.1 Autoclustering

In looking over the data we collected profiling the original matchmaking architecture, we noticed that the negotiator spent the majority of its time considering job ClassAds that did not result in a match. On average, just 10.0% of the ads considered during a cycle resulted in a match, which meant that nearly 90% of the negotiator's time was being spent doing work that did not produce a tangible result (i.e. a match). The numbers in the other pools we profiled, such as the GLOW pool, were similar — in GLOW, 15.9% of the ads considered resulted in a match.

We developed and implemented a technique termed *auto clustering*, which attempts to automatically cluster together job ClassAds that are identical for purposes of matchmaking. The approach is to assign each job an *autocluster id*, or signature. By definition, if a given job ClassAd with autocluster id X could not be matched, then none of the other jobs with auto cluster id X could be matched.

This effectiveness of this approach is dependent upon common usage of Condor. It was our hypothesis that when a typical Condor user submits hundreds of jobs, these jobs would normally differ in ways that are *not* significant for purposes of matching against a machine. In other words, we hoped that most job ClassAds from a given user differ in terms of input files, command line arguments, and output files, and generally have very little difference in terms of requirements or preferences upon a given machine.

Our algorithm for automatic determination of an auto cluster id is as follows:

1. At the start of a negotiation cycle, the negotiator will inspect each machine ClassAd and create a list of attribute names that cannot be resolved within the scope of the machine ad itself. In bi-lateral matching, it can be assumed that these undefined attributes will need to come from a job ClassAd. We call these undefined attributes *significant attributes*.

2. The negotiator adds to the set of significant attributes any unresolved attributes contained in negotiator policy expression when these expressions are evaluated within the context of typical machine ClassAd.

3. The negotiator passes the union of all significant attributes it found to each schedd.

4. For each job, the schedd augments the significant attributes received from the negotiator by adding the job's `Requirements` and `Rank` attributes, as well as any job ClassAd attribute that is referenced by the job's own `Requirements` of `Rank` expression. The value of all the job's significant attributes and their corresponding values are placed into a string and hashed. The hash value is said to be the job's *matchmaking signature*.

5. All jobs with the same signature are assigned the same unique autocluster id.

In order to enhance performance, the autocluster id for each job is cached by placing the id into the job ClassAd itself, along with a list of attributes that were used to compute the signature. This cached value is discarded and recomputed if the list of significant attributes received from the negotiator changes, or if any of the job attributes used to compute the signature change. Measurements regarding the expense of computing autocluster ids and therefore the value of this caching is presented in Section 6.

We next modified the negotiation process so the schedd will ignore for the remainder of the negotiation cycle any job ClassAd that has the same autocluster id as a job for which the negotiator failed to find a match. With the introduction of auto clustering, the average percent of job ads considered by the negotiator that resulted in a match jumped from 10.0% to 64.3% with the Comp Sci Pool trace data, and similarly went from 15.9% to 78.3% in the GLOW Pool.

4.2 Negotiator MatchList Caching

In the original matchmaking architecture, the negotiator receives a job ClassAd from a schedd and proceeds to pick a machine ClassAd by (i) evaluating the job and machine `Requirements` expression for each machine ad, discarding any ads that do not match, and then (ii) traversing the list of matched machine ads evaluating job and machine `Rank` expressions in order to pick the most preferred match. The time it takes for the negotiator to find a matching machine ad given given one job ad is mainly a function of the number of machine ads. In the Comp Sci Pool, which had an average of 1236 machine ads present during the week of our trace data, we were shocked to discover that the negotiator took an average of 0.30 seconds to find a match. The GLOW Pool, being larger with an average of 2205 machine ads, took long as expected with an average of 0.66 seconds per match.

In order to decrease the amount of time it takes to find a match per job ad, we introduced soft state into the negotiator. Previously the negotiator would perform a maximum found algorithm on the list of matching machine ads in order to return the highest ranking match. We modified the negotiator to instead perform a quicksort on the list of machine ads, return the machine ad from the top of the list, and then *cache this sorted list of machine ads*.

We took advantage of the autocluster id to know if the cache is valid. If the next job to arrive at the negotiator has the same autocluster id as the previous job considered, the negotiator can simply pop the next machine ad off the top of its cached results. If the next job to arrive has a different autocluster id, we invalidate the cache. We termed this technique *matchlist caching*. In the worse case scenario that every job ClassAd has a unique autocluster id, matchlist caching has a slight performance hit since we perform a quicksort ($\mathcal{O}(n \log n)$ on average) instead of a maximum found ($\mathcal{O}(n)$) algorithm upon the list of machine ads, but we have a big win on a cache hit.

With matchlist caching, the average time to find a match during the negotiation cycle dropped to 0.18 seconds and 0.34 seconds for the CS Pool and GLOW pool, respectively.

Note that our implementation for only caches one matchlist; we expect even better results if we kept multiple matchlist caches, one for each autocluster id encountered during a negotiation cycle.

4.3 Advertisement of Resource Requests

Our auto clustering and matchlist caching mechanisms help achieve our performance goals for this project, but the do not help the negotiator make better globally optimized decisions. To achieve this, we desire the negotiator to have efficient access to of the job ClassAds at once.

One simplistic approach would be to simply have all the schedd daemons forward all job ClassAds to the negotiator at the start of the negotiation cycle, and use caching techniques to minimize the necessity of such a bulk transfer. This approach has several failings. First of all, this would increase the memory footprint of the negotiator linearly with respect to the number of jobs submitted across all schedds. Such centralization of all job ClassAds also neutralizes the advantages of Condor's distributed approach between schedds and negotiators. Furthermore, job ClassAds normally contain nearly one hundred attributes about the job, including many attributes that are highly dynamic in nature (such as current cpu usage, memory usage, remote host where it last ran, etc). Such dynamic information could make caching approaches challenging.

Instead, we devised a mechanism that leverages autoclustering. The approach is to distill a queue of job ClassAds into a set of *resource request ads*. A resource request ad is a ClassAd that contains:

1. A set of ClassAd expressions that describe requirements and preferences on matching machine ads.
2. A resource request identifier.
3. A resource count that states how many of the described resources are desired.

Each schedd distills its own queue of jobs into a correspondingly smaller number of resource request ads. Each unique autocluster identified by each schedd results in a corresponding resource request ad. From the list above, item 1 is the group of significant attribute names and values that went into computing the signature hash for the autocluster, and item 2 is the autocluster id itself. Recall that the schedd is asynchronously advertising a *submitter ad* to the collector that contains the submitting user's name and a simple count of unmatched jobs. We changed the submitter ad to also contain, nested within, all of the resource request ads distilled from that user's job ad. This nesting was achieved by serializing each resource request ad into a string and then inserting the strings into the submitter ad. In effect, we changed the negotiation protocol from “do you have a match for this job? how about this job? and how about this job?” to instead be “give me *X* number of resources that look like this, and *Y* number of resources that look like that”.

This design also has the property of minimizing the memory footprint on the central manager, since we expect (i) the number of resource request ads to be much smaller than the number of job ads in the system, and (ii) each resource request ad will have far fewer attributes than a job ad since the resource request ad solely contains those attributes that are required for the matchmaking process. Job ad attributes that are not required for matchmaking, such as the attributes required to launch and track the job (such as environment variables, file path names, etc), are automatically kept decentralized.

By asynchronously advertising the resource requests, we further improve negotiation cycle performance by breaking up the serial nature of the original protocol. In Condor’s current design, the negotiator contacts the schedd and says “find all candidate jobs for user X, sort them, and send them to me one at a time”. When the queue has thousands of jobs, this operation can take many seconds because sorting the jobs may involve evaluating many arbitrary ClassAd expressions on top of the sort itself (in Condor, the priority of a job itself can be expressed as a ClassAd expression). Our design takes this step out of the negotiation cycle bottleneck; furthermore, it enables multiple schedds to perform this operation in parallel.

In our design, in addition to the schedd sending resource requests asynchronously, the negotiator also sends match information to the schedd asynchronously. When the negotiator makes a match, it sends the machine ClassAd along with resource request identifier from the corresponding resource request ClassAd to the schedd. The schedd uses the resource request identifier as an optimization hint; assuming it trusts the negotiator, the schedd knows this machine ad can be used to run any job in its queue that has the same autocluster id value. This prevents the schedd from having to iterate through its job queue to find a job that matches the machine ad supplied to it by the negotiator.

4.4 Protocol Walkthru

In summary, a negotiation cycle that has been enhanced with our resource request mechanism works as follows.

All schedds perform the following asynchronous operation periodically:

1. Fetch the list of significant attributes contained in the machine ads from the collector (because the negotiator no longer contacts the schedd except to send it results, the negotiator places this information into the negotiator daemon’s ClassAd that is published to the collector).
2. Compute (and cache) autocluster ids for all jobs in the queue.
3. For each user, sort all of the user’s unmatched job ads and extract an ordered list of autocluster ids.
4. For each autocluster id, create a resource request ad, and insert all resource request ads for the user into the user’s submitter ad. Publish the submitter ad to the collector in the usual manner (typically a UDP datagram).

When the negotiator performs a negotiation cycle, it does the following:

1. Fetch all machine and submitter ClassAds from the collector. All submitter ads have a sequence number created by the schedd; submitter ads (and thus their embedded resource request ads) are cached in the negotiator until either a lease expires, or a newer submitter ad with a larger sequence number is retrieved from the collector.
2. Traverse all machine ClassAds to derive job ad significant attributes (see Section 4.1 on page 4), and publish this information into the negotiator ClassAd.
3. Perform the exact same matchmaking algorithm as Condor has always done, but (i) instead of communicating with the schedds to fetch a job ClassAd, the negotiator consults lists of resource request ads, and (ii) use the matchlist caching mechanism as described in Section 4.2.
4. When a match is found, the negotiator sends the machine ClassAd and the corresponding resource request id to the schedd. It then decrements the count

attribute of the corresponding resource request ad cached in its RAM. When the count reaches zero, that resource request ad is ignored.

5 Implementation

The changes described in the proceeding section were implemented in C++ and incorporated into the pre-existing Condor code base (in the hope that they will all eventually make it into production, although time will tell here). The changes required about 2000 lines of code.

6 Evaluation

The success of our mechanisms, such as autoclustering, hinged on the characteristics of common workloads. It would have been trivial for us to create a synthetic workload to show best-case and worst case scenarios (in the case of the resource request mechanism, this could be a job queue with 10,000 job ads that distills down to either one resource request ad or 10,000 request ads, respectively). Therefore, our real interest was to observe the impact of our work in the real-world — with real-world job workloads and actual job and machine ClassAd policies.

Unfortunately, large production Condor pools are always changing; users are coming and going, jobs and machines are entering and leaving, and policy expressions are being changed almost daily to meet ongoing priority requirements. This presented a challenge, as we needed a controlled environment in order to evaluate multiple different mechanisms properly.

Our solution was to create software to enable Condor's collector and schedd daemons to accept trace files as input and run through just one complete negotiation cycle. The collector was modified to read a file containing a number of machine ClassAds at startup. For the schedd, we wrote a program that transformed a file containing job ClassAds into the transactional log file format used by the schedd for persistence; the schedd would read this log file upon start-up as usual to restore its state. Finally we modified the schedd to not attempt to contact an execute machine upon receiving a match.

To create the input trace files, we just invoked unmodified `condor_q` and `condor_status` client tools with appropriate flags to retrieve all of a pool's job and machine ClassAds, respectively. For nearly six days, every six hours we took a snapshot of the state of three busy production Condor pools on campus — the Comp Sci Dept pool, the Grid Lab of Wisconsin (GLOW) pool, and the Computer Aided Engineering (CAE) pool — for a total of 69 trace files. We felt that six days of traces from three actual pools subjected our algorithms to the necessary wide array of inputs.

We then started a schedd (injected with an entire pool's job ads) and negotiator together on the same machine, and took measurements as the system performed one negotiation cycle. The measurements we recorded were:

- the number of machine ads, the number of job ads, and the number of submitting users involved;
- the time it took the schedd to perform autoclustering of all jobs;
- the number of autoclusters created;
- the number of job ads considered that resulted in a match;
- the number of job ads considered that failed to produce a match;
- the time (in seconds) for the negotiator to fetch all the job and submitter ads from the collector;
- the time (in seconds) for the negotiator to perform one negotiation cycle;
- and the image size of the collector process, according to `/bin/ps`, at the conclusion of the negotiation cycle.

We ran each of the 69 trace files four times (by submitting our experimental jobs to the CS Dept Condor pool, of course!). For the first run, we established a baseline by disabling all of the mechanisms outlined in Section 4. For the second run, we enabled just the autoclustering mechanism. For the third run, we enabled autoclustering plus matchlist caching. For the fourth run, we enabled resource request advertisement (which implies autoclustering as well) and matchlist caching.

Overall, we were very pleased with the results. Figure 3 on the next page shows the impact of our work upon negotiation times using the traces from the Comp Sci Department pool. In that pool, on average our baseline negotiation cycle took 2583 seconds; with autoclustering, it

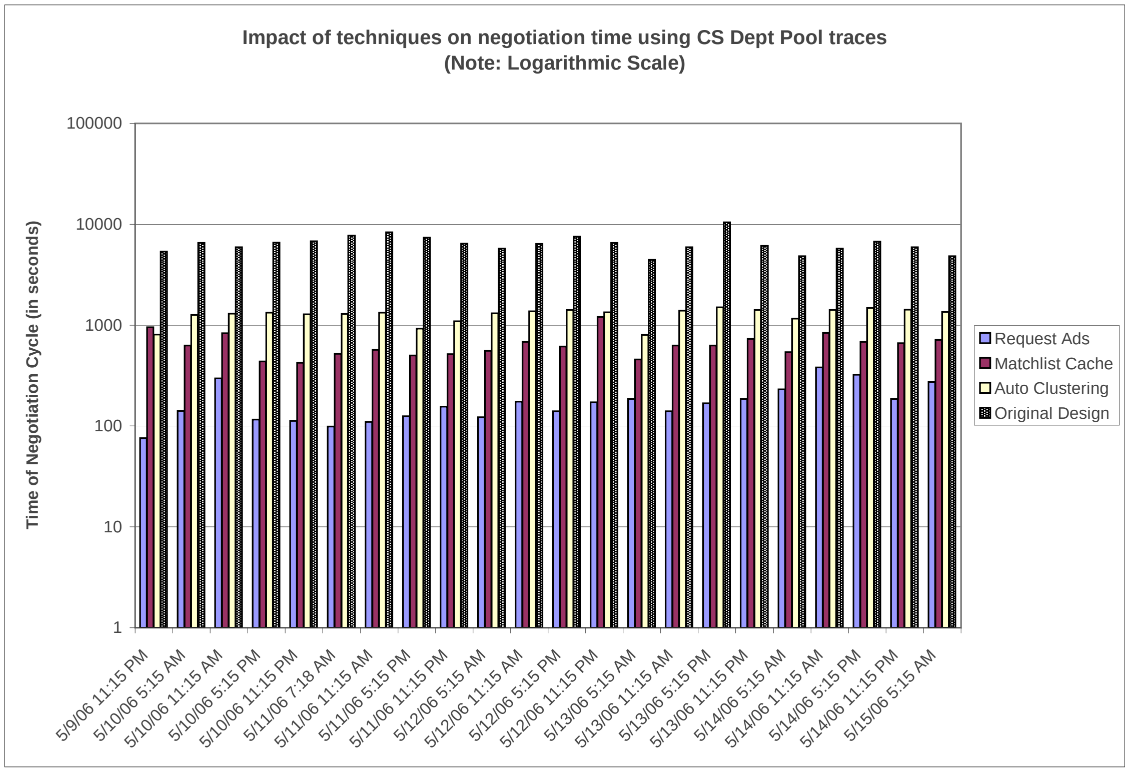

Figure 3: **Negotiation Cycle Performance.** *This chart shows the performance impact of the three techniques described in this paper, as well as the original (baseline) implementation.*

dropped to 366 seconds; adding in matchlist caching, it dropped to 223 seconds; and finally with resource request advertising added into the mix, the average dropped yet again to 129 seconds. We were able to meet our design goal of enabling the negotiator to have a centralized view of all job ClassAds, and still improve negotiation cycle performance by approximately 20x. The performance improvements in the GLOW and and CAE pool were similar. For instance, the average negotiation cycle time in the baseline case for GLOW was 6468 seconds; with resource request advertising this dropped to an average of 179 seconds.

The mechanism that produced the most dramatic results was autoclustering, which can be explained when we look at the data related to autoclustering. In the course of the six days within the Comp Sci pool, an average of 85 active users had 5831 jobs submitted. These 5831 job ClassAds were distilled down into an average of just 372 autoclusters (and thus 372 resource request ads) — an approximate reduction by 15x the number of ads the negotiator needed to consider during the matchmaking cycle. It took the schedd an average of 4.7 seconds to perform the autoclustering on all 5831 (on average) job ads, but recall the autocluster information is cached. Furthermore, the autoclustering load will be partitioned across multiple schedds.

One initial concern of centralizing the job information at the central manager was the impact on the memory footprint. In the Comp Sci pool, our baseline tests showed the maximum size of the central manager state at 29.0 MB. Our tests using the resource request advertisement showed the maximum at 29.8 MB. For the cost of less than 1MB of memory, the resource request algorithm allows the Condor negotiator to explore more efficient, globally optimized scheduling algorithms. This 800KB of memory is what it takes to store an average of 372 resource request ClassAds; each ad takes only about 2k because the number of significant attributes (average of 10) is much smaller than the number of attributes contained in a job ClassAd.

7 Conclusion

In conclusion, I had a lot of fun working on this project. It enabled me to explore areas of Condor that I have not been able to do so in the past, and also provided me with an opportunity to take measurements on the performance of Condor. I hold out hope that my work, in particular the resource request mechanism, will eventually make it into the publically released Condor. Even if it does, however, the experience I gained will stay with me, and the process I developed for feeding trace data to measure Condor's matchmaking performance will hopefully be useful to others on the project.

I enjoyed the class immensely (my first class in over 15 years!). Have a great summer!!

References

[1] James Frey, Todd Tannenbaum, Ian Foster, Miron Livny, and Steve Tuecke. Condor-G: A computation management agent for multi-institutional grids. In *Proceedings of the Tenth IEEE Symposium on High Performance Distributed Computing (HPDC)*, pages 7–9, San Francisco, California, August 2001.

[2] D. Gale and L.S. Shapley. College admissions and stability of marriage. *American Math. Monthly*, 69(1):9–15, 1962.

[3] Michael Litzkow, Miron Livny, and Matthew Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, June 1988.

[4] Miron Livny and Rajesh Raman. High-throughput resource management. In Ian Foster and Carl Kesselman, editors, *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1998.

[5] Rajesh Raman. *Matchmaking Frameworks for Distributed Resource Management*. PhD thesis, University of Wisconsin, October 2000.

[6] Rajesh Raman, Miron Livny, and Marvin Solomon. Matchmaking: Distributed resource management for high throughput computing. In *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing (HPDC7)*, Chicago, IL, July 1998.

[7] Alain Roy and Miron Livny. Condor and preemptive resume scheduling. In Jarek Nabrzyski, Jennifer M. Schopf, and Jan Weglarz, editors, *Grid Resource Management: State of the Art and Future Trends*, pages 135–144. Kluwer Academic Publisher, 2003.

[8] Douglas Thain, Todd Tannenbaum, and Miron Livny. Condor and the grid. In Fran Berman, Geoffrey Fox, and Tony Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley & Sons Inc., December 2002.