1 Introduction

The Condor high throughput distributed system is underpinned by the matchmaking process which performs distributed resource allocation. While the implementation of matchmaking is quite robust, the tools for identifying and explaining why a resource request does not find any matching resource offers are somewhat insufficient (see Figure 1). In order to provide a more detailed analysis of such a scenario, new code must be developed and new algorithms must be designed. This paper describes a research project with the intent of implementing robust matchmaking analysis and integrating it with the current Condor user tools.

The research I have conducted prior to the initiation of this project has been directed towards developing the algorithms necessary for matchmaking analysis, and implementing a prototype analysis tool in Java. The scope of this project consists of the implementation of matchmaking analysis in C++ and the integration of this code with current Condor tools also written in C/C++. Due to time restrictions and other commitments, only a subset of the algorithms present in the Java prototype were implemented in the C++ version.

2 The Project - A Tale of Two Ports

The long term goal of this project was to incorporate ClassAd analysis algorithms into the Condor user tools, in particular `condor_q`. The completion of this project rested on the completion of two orthogonal ports. The first was to port the Condor user tools (`condor_status` and `condor_q`) and the supporting infrastructure from old ClassAds to new ClassAds. Aside from a much cleaner implementation, new ClassAds offer extra added functionality that is integral to robust ClassAd analysis. The second port was to rewrite the existing Java implementation of ClassAd analysis in C++. This port was necessary for the integration of the Condor tools, the C++ ClassAd library, and the ClassAd analysis algorithms.

2.1 Porting Condor Tools to New ClassAds

The two Condor user tools selected for the port to new ClassAds were `condor_status` and `condor_q`. The former allows the user to get useful information from a Condor pool. While not directly relevant to matchmaking analysis, `condor_status` was a good place to start the port. More important is the latter, which allows the user to query the job queue. `condor_q` (with the `-analyze` option) is currently the only source of information about unmatched jobs. Typical feedback from `condor_q -analyze` can be seen in Figure 1. As stated before, the main motivation of this research is to provide a more detailed analysis of the unmatched job scenario.

```
27989.000:  Run analysis summary.  Of 715 resource offers,
          688 do not satisfy the request's constraints
           27 resource offer constraints are not satisfied by this request
            0 are serving equal or higher priority customers
            0 do not prefer this job
            0 cannot preempt because PREEMPTION_REQUIREMENTS are false
            0 are available to service your request
```

Figure 1: Typical output of *condor_q -analyze*

So called "new" ClassAds are a more sophisticated implementation of the ClassAd language, than the "old" ClassAds currently in use in the Condor v6.x releases. Firstly the ClassAd code is decoupled from its ancestor the "AttrList", which was a simple list of resource attributes. This decoupling allowed for the development of a new paradigm, in which a ClassAd could be contained in another ClassAd, providing the framework for multilateral matching. As a practical matter, and aside from the benefits to this project, porting the Condor user tools to new ClassAds is useful in two ways. All of Condor will have to be ported to new ClassAds eventually, so porting these tools will provide something of a head start. Also, the experiences gained from the port may be useful when the rest of Condor begins using new ClassAds.

The paradigm shift directly impacts this project in two important ways. One added functionality of the new ClassAd libraries is the ability to carry out a partial evaluation or "flattening" of a ClassAd expression. This feature is invaluable to ClassAd analysis, where a partial evaluation of a requirements expression results in an expression containing only externally defined variables. A second but less crucial feature of the new paradigm is the ability to nest ClassAds. This comes in useful for expressing results of an analysis in a hierarchical fashion.

The porting itself encompassed the modification of not only the front ends of the user tools, but all of the back end code that supports them. Though each piece of code required different modifications, the port could be summarized in four seperate parts:

1. *API Shift* - The simplest modifications were replacing deprecated old ClassAd methods with their new ClassAd equivalents. As an example the function call to `EvalString` had to be replaced with `EvaluateAttrString`.

2. *Matchmaking* - This was a slightly more complicated conversion, as the matchmaking framework in old ClassAds bears no resemblance to the new ClassAd equivalent. This also brought up an issue with the new implementation which uses nested ClassAds to build what is called a MatchClassAd. The structure of a MatchClassAd can be seen in Fiqure 2. The MatchClassAd deals with back compatibility in that references to `my` and `target` are equivalent to `self` and `other`. However, this structure does not support implicit attribute references to the target ClassAd such as `Arch == ''INTEL''` instead of `other.Arch == ''INTEL.''` One can explicitly set the parent scope of the ClassAd to be the target, and then vice versa for a symmetric match, but then the explicit attribute references are not supported. In the Java ClassAd implementation both implicit and explicit attribute references are supported in matchmaking. Perhaps the C++ version should adopt the Java version's framework.

3. *ClassAd Conversion on the Wire* - The C++ ClassAd API includes back compatibility methods `getOld-ClassAd` and `putOldClassAd` which convert server side old ClassAds to client side new ClassAds and vice versa. These methods replaced the old `get` and `put` methods from old ClassAds, but two unanticipated complications arose during the port. First of all, new functionality had to be added to the ClassAdUnparser to print the meta-equals and the meta-not-equals operations old ClassAds style: `=?=` and `=!=` as opposed to `is` and `isnt` to prevent the server side from choking upon reciept of a ClassAd. Secondly, a third method `getOldClassAdNoTypes` had to be implemented to replace the old AttrList get method used in `condor_q` which did not support ClassAd types.

4. *Updating Data Structures* - Two data structures, ClassAdList and AttrListPrintMask, were an integral part of the Condor user tools. The former is a list of ClassAds that supports sorting by attribute values, and the latter facilitates tabular printing of ClassAds. Both classes were heavily dependent on old ClassAd and AttrList methods. ClassAdList had to be completely rewritten as a wrapper for a ClassAd instantiation of the List template. AttrListPrintMask required fewer modifications, but for the sake of clarity its name was changed to ClassAdPrintMask.

2.2 Porting Java Code to C++ Code

As previously stated, only a subset of the Java code has been ported to C++. This code focuses on job ClassAd analysis at the granularity of atomic boolean expressions know as Conditions. The resulting feedback identifies which Conditions the user should keep, and which should be removed so that some subset of

```
[
  symmetricMatch   = leftMatchesRight && rightMatchesLeft;
  leftMatchesRight = adcr.ad.requirements;
  rightMatchesLeft = adcl.ad.requirements;
  leftRankValue    = adcl.ad.rank;
  rightRankValue   = adcr.ad.rank;
  adcl             =
      [
          other    = .adcr.ad;
          my       = ad;       // for condor backwards compatibility
          target   = other;    // for condor backwards compatibility
          ad       =
             [
                 // the ``left'' match candidate goes here
             ]
      ];
   adcr             =
      [
          other    = .adcl.ad;
          my       = ad;       // for condor backwards compatibility
          target   = other;    // for condor backwards compatibility
          ad       =
             [
                 // the ``right'' match candidate goes here
             ]
      ];
]
```

Figure 2: The structure of a *MatchClassAd*

the resource offers are accepted by the job Requirements expression. Though this is only a fraction of the functionality needed by a robust analysis tool, it still requires a fair amount of supporting code.

The following classes and structures have been implemented:

- *ResourceGroup* - A wrapper for a List of ClassAds representing resource offers. The level of abstraction exists to facilitate a smooth transition to ClassAd Collections.
- *BoolExpr* - A base class for ClassAd expressions which are expected to evaluate to a boolean value given a sufficient context.
  - *Condition* - A BoolExpr of the form ATTR OP VALUE.
  - *Profile* - A BoolExpr consisting of a conjunction of Conditions
  - *MultiProfile* - A BoolExpr consisting of a disjunction of Profiles
- *BoolVal* - An enum type representing literal values in the four valued logic of the ClassAd language.
- *BoolVector* - A wrapper for an array of BoolVals
  - *AnnotatedBoolVector* - A BoolVector with annotations indicating how many and which contexts (resource offers) it represents.
- *BoolTable* - A wrapper for a two dimentional array or table of BoolVals.
- *Explain* - A base class for a repository for results of an analysis. Subclasses ConditionExplain, ProfileExplain, and MultiProfileExplain are self-explanatory (so to speak).

- *Analysis* - Where everything comes together. A Library of methods which perform functions key to the overall analysis.

Aside from the renaming of a few classes, the above list corresponds to classes in the original Java code.

As expected, a number of issues came up during the implementation of these classes as there are fundamental differences between Java and C++. The availability of templates (namely the List template) was quite helpful and reduced the need for explicit casting that is so ubiquitous in Java. On the other hand, the abscence of exceptions required that most methods return a boolean value for failure/success. Also, passing objects as parameters was avoided if possible, and passing arrays as parameters was avoided completely. When necessary objects were passed by reference. Finally, in an attempt to avoid memory allocation in constructors, all classes were equipped with an `Init` method which handles memory allocation and returns a boolean success value.

2.3 Putting It All Together

Once the condor tools had been updated to use new ClassAds, and the infrastructure for job requirements analysis had been implemented in C++ it was time to bring the two pieces together. The entry point for the matchmaking analysis code is clearly `condor_q -analyze`. At this point in the `condor_q` code we have a ClassAdList containing all available resource offers and a single ClassAd representing the request. These two objects are passed to the analysis API.

Next, the offer list is converted to a ResourceGroup, and the Requirements expression from the request is flattened and converted to a MultiProfile. For each Profile in the MultiProfile a BoolTable was built with columns representing resource offer ClassAds, rows representing individual Conditions, and entries corresponding to evaluating the given Condition in the context of the given resource offer. The BoolTable is used to deduce the best suggestion to the user. When the ideal suggestion has been computed, the appropriate information is stored in the varous Explain objects. Finally the information in the Explain objects is presented in text form.

An example of the output is shown in Figure 3. While this example does not show off the full functionality of job requirements analysis, it underscores the further work that needs to be done. First of all it is clear that some of the Conditions are redundant, and secondly this example begs analysis of the Requirements expressions of the 27 machines that did match the job's constraints. This is only a sampling of the extensions that can be built on the foundations established by this project, further examples are discussed in the following section.

```
=====================
RESULTS OF ANALYSIS :
=====================

REQUIREMENTS: matched 27 of 704 offers
  Profile 1 matched 27 offers
    Memory > 64                matched 553  - suggestion: KEEP
    Arch == "INTEL"            matched 576  - suggestion: KEEP
    OpSys == "SOLARIS26"               matched 78  - suggestion: KEEP
    "INTEL" == Arch            matched 576  - suggestion: KEEP
    "SOLARIS26" == OpSys               matched 78  - suggestion: KEEP
    Disk >= 5332               matched 701  - suggestion: KEEP
=====================
```

Figure 3: Results of matchmaking analysis for the job in Figure 1

3 Future Work

Clearly much more work needs to be done with the ClassAd analysis code in order for it to be robust enough to be a useful extension to the current Condor user tools. First and foremost is the need for job attribute analysis (or resource offer requirements analysis depending on which way you look at it). Most of the required functionality for this has been implemented in the Java version, and key data structures (Intervals and HyperRectangles) have already been implemented in the C++ version for the purpose of more efficient matchmaking. Also quite key is the ability to detect contradictions and redundancies in Requirements expressions. Again much of this functionality exists in the C++ implementation, so it is only a matter of ferreting it out and adding any necessary enhancements.

Some more distant goals include analysis of arbitrarily structured molecular boolean expressions and more complex atomic boolean expressions such as comparisons and predicates. Other goals in the long term are dealing with nested ClassAds and multilateral matchmaking, and analyzing Rank and other numerical expressions. A number of enhancements may be made to improve the quality of the feedback of the user. In addition to suggesting removals of Conditions, it would be useful to suggest modifications, and identify contextual contradictions (such as a request for a SPARC architecture running WindowsNT). Support for a graphical user interface is also quite important, as is support for constructing expressions from Conditions and Profiles. Finally, there are plenty of opportunities for performance optimization and robust error propagation.

4 Conclusion

One conclusion that may be drawn from this project, and indeed any project involving a port from Java to C++ is that C++ is a great deal trickier to use than Java. This is not a novel proposition, but it is worth elaboration. While the Object Oriented model in C++ is quite similar to Java, the absence of pointers and explicit memory management in Java leads the programmer to overlook these issues when porting to C++. As described above the class structure of the Java prototype was maintained, but the implementation of these classes was somewhat different. So why use C++? The standard answer to this question is that it is faster and more widely used. The more meaningful answer is that to date most "real" application programming is done in C++. Java may have gained popularity – and with good reason: it is a very cleanly implemented and easy to use language – but it has not displaced C/C++ as the standard application programming language. That being said Java is quite useful as a prototyping tool, and as such was the right place to start implementing matchmaking analysis.

The experience of porting from old ClassAds to new ClassAds is more directly useful to the Condor project. The majority of the port as stated was adapting to a shift in the ClassAd API, but there are some issues with matchmaking that need to be addressed. As a side note, another porting will need to be done when the Condor code begins using ClassAd Collections. To begin with the ClassAdList class ought to be eliminated and replaced with a client side ClassAd Collection. The algorithms for matchmaking analysis and even matchmaking itself will have to be reexamined to determined if ClassAd Collections can offer any enhancements in performance.

Finally it should be noted that both ports were completed in a relatively short period of time. Now that the basic infrastructure for matchmaking analysis is in place, it should be quite feasible to port the remainder of the Java prototype to C++, and to implement additional functionality as well. The "new" ClassAd C++ API is quite robust and lends itself well to matchmaking analysis. What's more, a number of bugs in the C++ API were uncovered in the process of this project. Further work in this area will have the added benefit of rigorously testing the more experimental code in the new ClassAd implementation, in addition to providing a robust and useful matchmaking analysis framework for Condor.