Lecture 5: Condor

Condor

a. QUESTIONS FROM REVIEWS:
   i. How would it work for data intensive computing? Is scavenging local workstations usable then?
   ii. What about power consumption of using idle workstations?
   iii. Is Parrot, using debugging interface to trap system calls and remote them, inefficient?
      1. In any case must block to do system calls
   iv. Checkpoint system only works with some programs (need to be linked against condor library)
      1. So?
   v. Todd Frederick: even with the existing safeguards, poorly or even maliciously crafted jobs can still cause minor headache for resource users. The very nature of cycle scavenging can incentivize resource owners to configure Condor to not serve the best interests of actual resource users, increasing the likelihood that background jobs will cause problems in subtle yet noticeable ways.
   vi. Single centralized matchmaker: is it a single point of failure?
      1. But show you can have multiple pools each with a matchmaker
   vii. How compare with VM based clusters?
      1. Condor has been extended to run virtual machines
   viii. How handle node failure?
   ix. What about jobs that don't use standard universe, such as pre-canned software that cannot be relinked or recompiled?
b. QUESTION: how is Condor different than Google, LARD, Inktomi?
   i. Goal: high-throughput computing, not arbitrary processes
      1. Means latency to start not important
      2. Means reliability of long-running jobs is important
   ii. Goal: started off in a heterogeneous environment
      1. No central authority
      2. Federate resources of multiple machine owners
      3. Has explicit matchmaking rather than central scheduling just placing jobs
   iii. Goal: reliability
      1. Want to submit job and have confidence that it will execute
         a. Restarts failed jobs
      2. Don't want to lose work if user comes back (via checkpoint); sprite might fail to migrate a process back
   iv. NOTE: doesn't promise to be a whole OS, just a cpu-intensive (and later data-intensive) execution engine for long-running jobs
      1. Not good for short jobs – delay of matchmaking too long
      2. Note: most useful systems are not fully general
   v. QUESTION: is being high-throughput a goal for Condor, or the only thing it is good at?

vi. QUESTION: How does assuming contribution of machines or federations of machine affect design?
    1. Cannot rely on Google-like assumptions of clusters of heterogenous machines under a single administrator
    2. Relies on users/machine owners to make decentralized policy decisions
    3. Provides best-effort service
    4. Has features to support what ever existing systems/protocols are in use to be as widely useful as possible.

c. QUESTION: What are key Condor entities:
    i. User: submits tasks to local queue on their computer
    ii. Problem solver: decides what jobs to submit and when
        1. If user has static set of work to be done, not control or data flow between jobs, then not needed
    iii. Agent (schedd): proxy for user in finding a place to execute jobs
        1. Submits request for resources every 5 minutes
    iv. Resource (startd): proxy for a machine, looking for jobs to execute
        1. Generally resources only run one job at a time, or one per CPU
        2. Submits classad every 5 minutes
    v. Matchmaker: scheduling agent to look for appropriate places to execute jobs
        1. ClassAds considered soft state – need to be resubmitted
        2. May preempt jobs on a machine if a better job comes about (ranked higher by machine)
        3. NOTE: could replicate (have multiple copies) or distribute (submit jobs to some, resources to some) if have enough of each.
    vi. Shadow: proxy for user to provide environment, access to files
        1. Runs near submission machine
        2. Solves heterogeneity, as it provides the standard environment
    vii. Sandbox: execution container on a machine, limits access and forwards requests to shadow

d. QUESTION: How does condor division of responsibilities make it scale?
    i. Assures an entity takes end-to-end responsibility for one aspect of a problem, rather than centralizing that responsibility.
        1. Agent: end-to-end reliable job execution
        2. Resource: end-to-end job seeking,
    ii. Allows federation of heterogeneous machines through matchmaking
        1. Not assume a single policy engine can handle all constraints; instead use flexible "ClassAds"

e. QUESTION: What is Condor take on transparency?
    i. Tries to be fairly compatible, but not 100 percent
        1. E.g. no IPC supported for checkpoint/migration
        2. Need to link to Condor I/O library, not use native system calls

f. Condor Scheduling Policy

i. QUESTION: it separates planning and scheduling. What does this mean?
    1. Planning: acquiring resources.
        a. Condor job classAd just asks for resources, doesn't say what the specific task is
        b. Can ask for multiple processors simultaneously for parallel jobs
    2. Scheduling: once have a place to execute, decide what to execute when
        a. If just a list of similar jobs, submit in any order
        b. Else **problem solver** runs sub-parts of a task in the required order on the available resources
            i. E.g. master/slave
            ii. Producer/consumer
            iii. Etc.
ii. How help?
    1. Separate resource allocation from scheduling as two separate problems
    2. Doesn't try for globally optimal solution, because jobs from different users are independent
    3. Allows users to specify their own goals for their programs once they have resources, without needing a global way to specify that
        a. E.g. DAG & master/worker problem solvers
iii. How handle deciding when a resource is available?
    1. Give it out for an extended period, so can have a little slop between jobs
iv. Supports **preemptive resume scheduling**
    1. Just like normal CPU scheduling
    2. Can schedule a job, then checkpoint it and preempt it for a higher priority job, such as the machine owner
    3. Can later resume the schedule
    4. Allows use of sporadically available resources

g. ClassAd mechanism
    i. QUESTION: Why is this needed? Is it really needed?
        1. ANSWER: if you have heterogeneity of any sort
            a. Different amount of memory, speed of CPU, class of users
        2. ANSWER: allows local, variable policies from different pool owners
    ii. QUESTION: Is it needed in a cloud environment such as Amazon Web Services?
        1. ANSWER: They have different classes of machines, users, need to figure out where to schedule
        2. ANSWER: May not need as rich a mechanism if you have less heterogeneity
    iii. QUESTION: how handle schema across multiple organizations
        1. Each might choose their own attributes (e.g. even language)

2.

iv. Process:
    1. Advertise resources, jobs
    2. Matchmaker looks for matches, notifies resource/agent
    3. Claiming step to verify match works

v. Example

Job ClassAd	Machine ClassAd
[	[
MyType = "Job"	**MyType** = "Machine"
TargetType = "Machine"	**TargetType** = "Job"
Requirements =	**Machine** = "nostos.cs.wisc.edu"
((other.Arch=="INTEL" &&	**Requirements** =
other.OpSys=="LINUX")	(LoadAvg <= 0.300000) &&
&& other.Disk > my.DiskUsage)	(KeyboardIdle > (15 * 60))
Rank = (Memory * 10000) + KFlops	**Rank** = other.Department==self.Department
Cmd = "/home/tannenba/bin/sim-exe"	**Arch** = "INTEL"
Department = "CompSci"	**OpSys** = "LINUX"
Owner = "tannenba"	**Disk** = 3076076
DiskUsage = 6000	]
]	

Figure 12. **Two Sample ClassAds from Condor.**

    1. Machines check whether they can run a job every two minutes
        a. Limits utility for short jobs
        b. Machines can have a local queue of jobs to run from clients they have accepted

vi. QUESTION: Why claim?
    1. Allows matchmaker to be stateless, as not need to know whether match worked
    2. Allows for conditions to have changed since advertisement (weak consistency)
    3. Allows for further conditions not expressible through ClassAds

vii. QUESTION: Where does policy about who to schedule where live?
    1. In the REQUIRES and RANK function: it selects what jobs/resources are appropriate, and which is best match

viii. How often does this run?
    1. Originally every two minutes

ix. QUESTION: Is matchmaker a bottleneck/single point of failure?
    1. Jobs are long-latency, so need not run continuously
    2. If store data on disk, can survive crash and restart without too much impact
    3. Could store data on a file server and have a copy that takes over if it fails …

h. Residual dependencies
    i. Condor is built around split execution: run jobs remotely, but send some I/O requests back to submitting node to get environment
    ii. QUESTION: is this a problem of residual dependencies?

1. Yes, but acceptable to provide uniform environment on heterogeneous system
2. Limited as to what is provided; not perfectly transparent

iii. QUESTION: Is sending I/O back to host a problem because it may cause failures?
1. ANSWER: could be, but allows more transparency. Chance of a single machine failure is fairly low, while failure of one of N machines is high. Plus, user is aware if their machine fails and will restart it?

i. Scalability concerns
i. Condor scalability is different than the other systems we looked at
1. Condor focuses more on scaling via aggregating pools of resources, rather than just making one pool bigger
2. Many mechanisms to allow one pool to submit to another (e.g. gateway flocking)
3. QUESTION: How does this occur outside of Condor?

ii. What is the major challenge for Condor in scaling outside of a single organization?
1. How do you set policy?
2. How do you implement communication to keep It simple, minimum number of parties involved
3. Who handles finding resources to solve a job across multiple organizations?
a. The submitter (direct flocking)
b. A gateway (gateway flocking)
c. Multiple queue managers (GRAM)
i. Then who chooses queues? What if you have too many or too few?
d. Separate resource allocation from scheduling jobs (Condor –G)
i. First acquire resources through GRAM
ii. Then use own scheduler to run jobs on a private pool

j. Reliability concerns:
i. How can you provide reliable execution of jobs?
ii. What failures can occur?
1. Central condor failure (e.g. services fail)
2. Job fails
3. Worker node failure (environmental failure)

iii. QUESTION: What do you need to do?
1. Detect whether failure was a failure of the job (it crashed) or the environment (node/OS crash)
2. If job crashed, did it happen because of the environment?
a. Would it work on another machine?

        b. If so, send somewhere else
    iv. QUESTION: How do you respond to failure?
        1. Brittle processes: just fail/stop immediately
        2. Resilient processes: handle failures and recover via logging, retry, reset, notify user/admin
        3. NOTE: brittle processes easier to write, try to use as much as possible
            a. E.g. starter, shadow: these are single instance per job
            b. Have a few well-crafted resilient processes
                i. Agent (schedd on client), resource (startd on a computer)
        4. Cleanup left to resilient processes – completely wipe out state of brittle process and restart

2. Condor Summary
    a. NOTE: Very successful, used widely here, several companies, CERN for LHC
    b. WHY?
        i. Addresses all the problems that come up in using workstation/clusters of computing
            1. Federating separate pool
            2. Policy of who can use machines and for what
        ii. Execution model simple enough
            1. Runs code without much change, but doesn't bend over backwards
        iii. Reliable
            1. Can depend on it to actually run your jobs with high probability
    c. Compare to other things we've seen?
        i. Focus on a specific aspect
        ii. Built for scale, heterogeneity, multiple policy domains