PonD : Dynamic Creation of HTC Pool on Demand Using a Decentralized Resource Discovery System

Kyungyong Lee
University of Florida
ACIS Lab. Dept of ECE.
klee@acis.ufl.edu

David Wolinsky
Yale University
Computer Science Dept.
david.wolinsky@yale.edu

Renato Figueiredo
University of Florida
ACIS Lab. Dept of ECE.
renato@acis.ufl.edu

ABSTRACT

High Throughput Computing (HTC) platforms aggregate heterogeneous resources to provide vast amounts of computing power over a long period of time. Typical HTC systems, such as Condor and BOINC, rely on central managers for resource discovery and scheduling. While this approach simplifies deployment, it requires careful system configuration and management to ensure high availability and scalability. In this paper, we present a novel approach that integrates a self-organizing P2P overlay for scalable and timely discovery of resources with unmodified client/server job scheduling middleware in order to create HTC virtual resource Pools on Demand (PonD). This approach decouples resource discovery and scheduling from job execution/monitoring — a job submission dynamically generates an HTC platform based upon resources discovered through match-making from a large “sea” of resources in the P2P overlay and forms a “PonD” capable of leveraging unmodified HTC middleware for job execution and monitoring. We show that job scheduling time of our approach scales with $O(\log N)$, where N is the number of resources in a pool, through first-order analytical models and large-scale simulation results. To verify the practicality of PonD, we have implemented a prototype using Condor (called C-PonD), a structured P2P overlay, and a PonD creation module. Experimental results with the prototype in two WAN environments (PlanetLab and the FutureGrid cloud computing testbed) demonstrates the utility of C-PonD as a HTC approach without relying on a central repository for maintaining all resource information. Though the prototype is based on Condor, the decoupled nature of the system components - decentralized resource discovery, PonD creation, job execution/monitoring - is generally applicable to other grid computing middleware systems.

Keywords

Resource discovery; self-configuration; P2P; virtual resources; high-throughput computing

Categories and Subject Descriptors

C.2.4 [**COMPUTER-COMMUNICATION NETWORKS**]: Distributed Systems

1. INTRODUCTION

High Throughput Computing (HTC) refers to computing environments that deliver vast amounts of processing capacities over a non-negligible period of time (e.g., days, months, years). This continuous computing throughput is a vital factor in solving advanced computational problems for scientists and researchers. Widely-used HTC middleware such as Condor [1] and BOINC [2] enable sharing of commodity resources connected over a local or wide-area network. Commonly, these systems use central managers to aggregate resource information and to schedule tasks. While this approach provides a straightforward means to discover available resources and schedule jobs, it can impose administrative overheads and scalability constraints. Namely, the central manager needs to be closely administered, and the failure of the manager node can render its managed resources unavailable for scheduling new tasks.

The growing adoption of on-demand computing-as-a-service cloud provisioning models and a large number of resources in data centers motivate the need for solutions that scale gracefully and tolerate failures, while limiting management overheads. These traits are commonly found in peer-to-peer (P2P) systems; nevertheless, the current prevailing approaches for job scheduling rely on specialized configuration of responsibilities for resources, e.g. for information collection and job match-making. With proper configuration, such approaches have been demonstrated to scale to several thousands of resources.

Scalability limitations of HTC middleware beyond current targets have been addressed in the literature. Raicu et. al. [3] addressed the inherent scalability constraint in existing HTC schedulers by proposing a fast and lightweight task execution framework which demonstrated improved scalability in a pool of tens of thousands resources processing millions of tasks. Sonmez et. al [4] measured the workflow scheduling performance in multi-cluster grids in simulated and real environments. The real-environment experiment was executed on the DAS-3 (a multi-cluster grid system in the Netherlands) revealing that limited capabilities of head-nodes result in overall system performance degradation as workflow size increases.

In this paper, we propose a novel HTC middleware approach, PonD, which leverages a scalable and flexible P2P decentralized resource discovery system to form a job execution Pool-on-Demand (PonD) dynamically from a large-scale resource pool. Within a PonD, unmodified client/server HTC middleware modules are self-configured on-demand, and take over the responsibility for job execution and monitoring. A key contribution of this paper, which differentiates PonD from related work, is the integration of decentralized resource discovery with a traditional centralized job-scheduling approach supporting existing job schedulers. By leveraging decentralized resource discovery during scheduling, we avoid the need for a central aggregator of information, and by using a centralized scheduler we can reuse existing HTC middleware. In addition to the enhancement on the scheduling scalability, this approach is able to provide improvements on the fault-tolerance, and combined with virtualization and infrastructure-as-a-service (IaaS) technologies, the PonD approach allows increased flexibility of scaling up/down virtual resource pools.

Our approach supports a large scale P2P overlay consisting of various candidate resources capable of joining a PonD. As typical in P2P systems, each resource, by default, can take the role of both job submitter and worker. Thus all members in a PonD, or connected resources, have the ability to dynamically create, join, and leave a PonD based on their queued-job demands. When creating a PonD, the requirements for the job are passed to the decentralized resource discovery module. This module forms a query and distributes it through a self-organizing, distributed query tree, which in turn aggregates the results in order to find resources which satisfy the job's requirements. The resource discovery module invites a portion of the usable resources to join a PonD owned by the job's submitter. Each resource handles the request by automatically configuring the HTC middleware to treat the job owner as the centralized manager, after which jobs are immediately scheduled and run.

Figure 1 illustrates the overall operation of PonD. Node A has a list of jobs in its job queue, and it sends a resource discovery query with job requirements. Using the query result, node A sends an invitation to nodes that satisfy ClassAd requirements to join its PonD — node A becomes the central manager node for its own PonD. Then, job scheduling and execution are processed by underlying HTC middleware. After job completion, Node A releases workers back into the larger resource pool, and returns to its initial status. In an abstract view, participating nodes exist in a large-scale, loosely-coupled resource "sea" — for instance, a wide-area opportunistic computing Desktop Grid, or a system that federates across multiple public/private clouds. When a node needs to run a job, it discovers and invites worker nodes to join a "pond" that is much smaller — for instance, a virtual cluster with hundreds of nodes aggregated into a single virtual resource pool. Upon job completion, nodes return to the "sea" and await other invitations or initiate their own PonDs. The discovery process is non-binding, and nodes are able to decide to accept or decline membership on a PonD according to local policies; once bound to a PonD, scheduling policies of the HTC middleware deployed apply.

The practicality in our work derives largely from our experiences with a Condor-based PonD implementation, C-PonD. We present both the construction of C-PonD as well as experimental results from analysis, simulation, and deployment on PlanetLab [5] and the FutureGrid distributed cloud computing infrastructure [6]. The simulation results verify the scalability of PonDs — specifically, pool creation

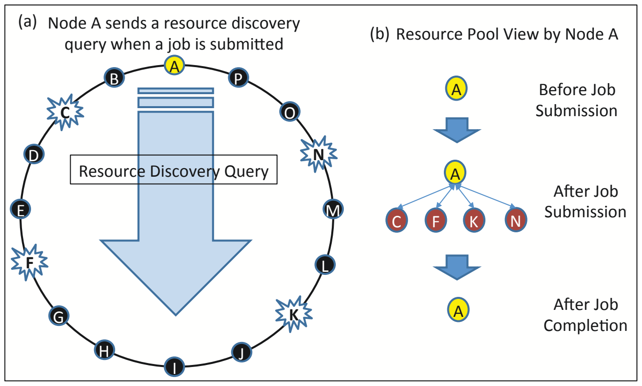


Figure 1: On-Demand pool creation when a job is submitted: (a)Node A sends a resource discovery query when it has a idle job in its job queue. (b)Based on the query result, Node A forms a condor pool with job requirement satisfying nodes (Node C, F, K, and N). After a job completion, worker nodes are released.

time grows as a function of $O(\log N)$, where N is the number of resources. The simulation results also evaluate the scheduling result correctness in case dynamically changing attributes are queried or participating nodes have unstable uptime. Experiments using FutureGrid resources capture a cluster/cloud scenario, and an experimental deployment on PlanetLab highlights a Desktop Grid scenario to demonstrate the feasibility of C-PonD on environments where resources are shared and widely distributed.

In summary, the main contributions of this work are:

- We propose a novel HTC system which integrates resource discovery with flexible query/resource representation in a decentralized query processing fashion based on P2P multicast trees, and dynamic middleware configuration to create resource pools on demand.
- We demonstrate this system by reusing existing, *unmodified* HTC middleware through a prototype implementation. It supports rich query processing and on-demand creation/tear-down of Condor pools.
- We evaluate the system from the perspectives of asymptotic scalability and its ability to provide timely resource discovery results through simulation-based analyses at large scales, and of its practical feasibility through experiments with smaller scale deployments of the prototype in WAN testbeds (FutureGrid and PlanetLab).

2. BACKGROUND

Focusing on Condor PonDs, in this section we give a brief description of its building blocks: Condor [1], a structured P2P network overlay, and a self-organizing multicast-tree build on a P2P network.

2.1 Condor

Condor [1] is a high-throughput computing framework that harnesses both dedicated and non-dedicated computing resources to create a resource pool. After gathering submitted jobs and available resource information from participating nodes, Condor negotiates and schedules inactive jobs to idle machines. Condor also provides Classified Advertisements (ClassAds) [7] that support a flexible and expressive way to describe job characteristics and requirements as well as resource information in order to determine matches.

In Condor, nodes can play the role of central managers, worker nodes, or job submitters. Roles are determined by running the appropriate daemons (i.e., *condor_startd, condor_schedd, condor_collector*, and *condor_negotiator*), and a single node may assume multiple roles.

condor_startd: This daemon is responsible for advertising resource capabilities and policies expressed using ClassAds to the *condor_collector* daemon of central manager as well as receiving and executing tasks delegated by the central manager. A node that runs this daemon is regarded as a worker node.

condor_schedd: A node which wants to submit a job runs this daemon. *condor_schedd* is in charge of storing user-submitted jobs at a local queue and sending resource requests to a *condor_collector* at a central manager node.

condor_collector: The role of this daemon is collecting resource information expressed using ClassAds and a list of *condor_schedd* with inactive jobs in their queue. A *condor_startd* periodically updates resource information ClassAds to their *condor_collector*. *condor_collector* also retrieves a list of inactive jobs from *condor_schedd* daemons at the time of negotiation.

condor_negotiator: This daemon is responsible for matchmaking between inactive jobs and idle resources. At each negotiation cycle, this daemon retrieves a list of inactive jobs and resource ClassAds from *condor_collector* daemon.

Usually, a condor pool consists of a single central manager that runs both an instance of the *condor_collector* and *condor_negotiator* daemons along with many worker nodes and submitters which run *condor_startd* and *condor_schedd* daemons, respectively. In order to provide reliable services in case of central manager node failure, fail-over central managers can be deployed using the high availability daemon.

In order to enable cross-domain resource sharing, Condor has two extensions, Condor-Flock [8] and Condor-G [9]. With these approaches, each node maintains a pre-configured list of other pools' central managers. Each central manager node also has to keep a list of allowed remote submitters. To overcome the pre-configuration requirements of condor flocking, Butt et al. [10] presented a self-organizing Condor flock that was built upon a structured P2P network. A locality-aware P2P pool is formed with central managers of each condor pool. When a central manager has available resources, it announces to other managers via the P2P overlay. When a central manager requires additional resources, it checks announcement messages from other central managers to determine where available resources are located. While our approach shares similar features to self-organizing Condor-flock [10] (self-configuration, and dynamic resource instantiation in response to jobs in the queue), the core mechanism for discovering nodes and self-organizing pools based on unstructured P2P queries with a logarithmically increasing overhead is a key differentiating factor.

Condor Glide-in [11] also provides a way to share idle resources across different administrative domains without a pre-configured list of other pools' central managers. When jobs are waiting in a queue of the *condor_schedd* daemon, *pilots* are created to find available resources for execution. In order to find available resources, a Glidein Factory manages a list of available resources across multiple Condor pools.

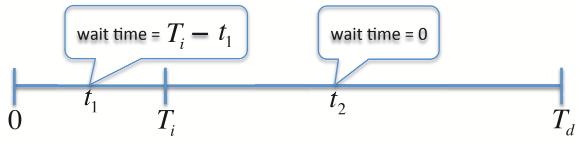


Figure 2: Condor negotiation procedure. T_d means the default scheduling period, and T_i means the minimum inter-scheduling time. A job submitted at t_1, where $t_1 < T_i$, has to wait $T_i - t_1$ for matchmaking.

Though the approach of flocking provides opportunities to share resources across different domains with bounded management overheads at a central manager, the cost of sequentially traversing pools in a flock list introduces non-negligible additional scheduling latencies at scale. Condor maintains a default negotiation period, T_d, and a minimum inter-negotiation time, T_i where $T_d \geq T_i$, as system configurations. Upon job submission, the negotiator checks if T_i has elapsed since the last negotiation cycle. If so, it starts a new negotiation cycle immediately. Otherwise, it waits until T_i elapses. The procedure is explained in Figure 2. In order to analyze the effect of the minimum inter-negotiation time to the overall waiting time in a queue until a matchmaking, the expected waiting time can be computed as follows.

$$\begin{aligned} E(T_{wait}) &= \frac{1}{T_d}\int_0^{T_d} (T_{wait}\ at\ time\ t)\,dt \\ &= \frac{1}{T_d} \times (\int_0^{T_i} (T_i - t)\,dt + \int_{T_i}^{T_d} 0\,dt) \qquad (1) \\ &= \frac{{T_i}^2}{2 \times T_d} \qquad (2) \end{aligned}$$

Equation 1 describes the waiting time of a job upon arrival at time t that can be explained with Figure 2. In the default Condor setup, T_d is 60 seconds and T_i is 20 seconds. Thus, the expected waiting time at each central manager during a flocking procedure is 3.3 seconds. Considering the sequential traversal of central managers during flocking, the waiting time increases linearly.

2.2 P2P system

P2P systems can provide a scalable, self-organizing, and fault-tolerant service by eliminating the distinction between service provider and consumer. C-PonD uses a structured P2P network, Brunet [12], which implements the Kleinberg small-world network [13]. Each Brunet node maintains two types of connections: (1) a constant number of near connections to its nearest left and right neighbors on a ring using the Euclidian distance in a 160-bit node ID space, and (2) approximately $log(N)$ (where N is the number of nodes) far connections to random nodes on a ring, such that the routing cost is $O(log(N))$. The overlay uses recursive greedy routing in order to deliver messages.

2.3 Tree-based Multicast on P2P

By leveraging existing connections in a structured P2P system, Vishnevsky et. al [14] presented a means to create an efficient distribution of messages in Chord and Pastry using a self-organizing tree. Each node recursively partitions responsible multicast regions by using the routing information at each node. DeeToo [15] presents a similar tree creation method on a small-world style P2P network, which is leveraged in PonD for decentralized resource discovery.

In a recursive-partitioning tree, each node is allocated a sub-region of the P2P ring over which to disseminate a message. A node then redistributes the message to neighboring nodes inside this region allocating new sub-regions to them. This process continues until the message is disseminated over the entire sub-region. For example, if a node, n_0, is assigned [*begin*, *end*] region for message distribution, the node checks its routing table to get a list of neighboring nodes that exist in [*begin*, *end*]. Assuming that node n_0 has neighboring nodes n_1, n_2, ..., n_{i-1}, n_i within $[begin, end]$, n_0 assigns sub-region $[begin,\ n_2)$, $[n_2,\ n_3)$, ..., $[n_{i-1},\ n_i)$, $[n_i,\ end]$ to nodes n_1, n_2, ..., n_{i-1}, n_i, respectively. These steps continue recursively until reaching leaf nodes that have no neighbors in the allocated sub-region.

In comparison to statically-built trees [16–18], the dynamic and self-organizing tree provides prompt responsiveness to node failures without the additional cost of maintaining pointers for children and parent nodes. Furthermore, a message does not have to be delivered to the static root node in order to be propagated to the entire set of resources.

3. ARCHITECTURE AND DESIGN

This section details the system design and the techniques used in resource discovery and self-organization of PonDs.

3.1 Decentralized resource discovery

Decentralized resource discovery in PonD shares similar motivations from related works of resource discovery in P2P [10, 19–23] but follows a fundamentally different approach. In a practical HTC system, the number of resource attributes can be large; for instance, default Condor installations have several dozens of attributes, and furthermore, users can provide their own attributes. Queries can be complex and include combinations of exact, ranges, and regular-expression matching, thus making resource discovery based on DHT key/value lookups or range-based queries insufficient.

Modifying the middleware or imposing limits to its utility is not a viable approach either. The approach taken by PonD ensures that unmodified HTC middleware can be used seamlessly. We leverage Condor ClassAds [7] and its match-making library for distributed query processing during discovery. By doing so, our decentralized resource discovery module inherits most of ClassAd characteristics, e.g. supporting range queries and regular expressions. For query dissemination and result aggregation, we use a self-organizing multicast tree. This approach can be applied to other structured P2P methods (e.g., Chord and Pastry) that support scalable multicasting with different resource representations and match-making engines (e.g., RDF and semantic queries).

3.1.1 Matchmaking module

The matchmaking module is responsible for matching requirements of jobs and resources by ranking nodes based upon their ability to satisfy a query. In order to obtain and distribute resource information, each resource uses *condor_startd* daemon which produces ClassAds. For match-making, requirements for desired resources and rank criteria are delivered as arguments. The following example illustrates arguments based on ClassAd syntax.[1]

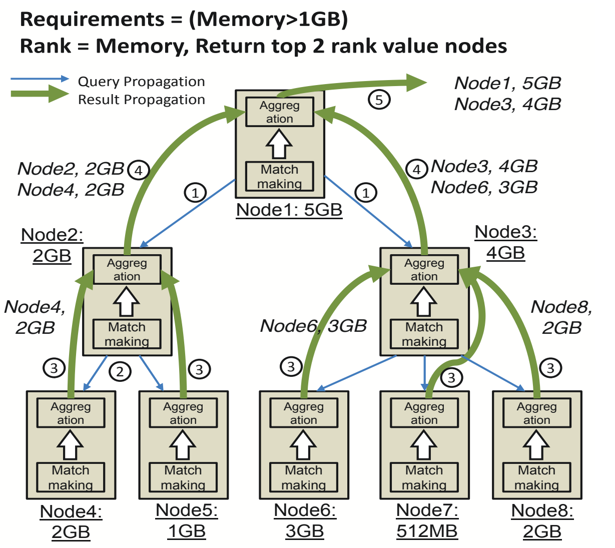


Figure 3: Resource discovery example. *Node1* wants to find the top two available Memory machines whose memory is greater than 1GB. *NodeID:Memory Size* value means match-making result, and <u>NodeID:Memory Size</u> value means available resource status.

Requirement=Memory>1GB && SWInstalled.Has ("Matlab") && regexp("*.edu",Hostname)

Rank = (Memory) + (KeyboardIdle*10)

The **Requirement** says the target resource's memory has to be greater than 1GB. The target machine also has to have *Matlab* installed, and the hostname of target machine has to end with ".edu". If a node satisfies the *Requirement*, it calculates a rank value. **Rank** is used to determine optimal candidates. This approach allows users to have the flexibility to specify their own job requirements and rank values.

3.1.2 Aggregation module

Upon completing the processing of a match-making task, a node returns the result to its aggregation module. The local aggregator waits for results from child nodes to aggregate its own and children nodes' matchmaking results. The aggregation is executed by sorting the rank value of query satisfying nodes and extracting the top ranked value nodes. After completing aggregation, the node returns the result to the parent node's aggregation function.

As with matchmaking, aggregation is processed independently at each node in a self-organizing multicast tree, and results are propagated back through the tree. This hierarchical information aggregation method provides scalability in a distributed information management system. The parallel processing of match-making and aggregation module is comparable to MapReduce [24] distributed computing: the match-making process is akin to a Map task, and result aggregation is akin to a Reduce task [25].

Figure 3 shows an example of the decentralized resource

[1]In the example, we modified ClassAd syntax for readability.

discovery. NodeID:Memory Size (underline) is the current resource status of each node, and *NodeID:Memory Size* (italic) is an aggregation result. Node1 (the root node) initiates a resource discovery query by specifying a requirement: e.g. a node's memory should be larger than 1GB. Nodes satisfying the requirement are ordered based on Memory size, and the two with the largest sizes are returned. The narrow line shows query propagation using a self-organizing multicast tree. The Aggregation module orders child nodes' result and returns a list of required number of nodes (thick line).

3.1.3 First-Fit Mode

In order to maximize the efficiency of aggregation and to discover the list of nodes that maximize rank, a parent node in a recursive-partitioning tree waits until all child nodes' aggregated results are returned. However, if it is acceptable to execute tasks on resources that satisfy the requirement but might not be the best-ranked ones, the query response time can be improved by executing a resource discovery query in *First-Fit* mode. With First-Fit, an intermediate node in a tree can return an aggregated result to its parent node as soon as the number of discovered nodes are larger than the number of requested nodes.

3.1.4 Redundant Topology to Improve Fault-Tolerance

While a tree architecture provides a dynamic and scalable basis for a parallel query distribution and result aggregation, it might suffer from result incompleteness due to internal node failures during an aggregation task. For instance, in Figure 3, the failure of *Node 3* in the middle of query processing would result in the loss of the query results for nodes *6*, *7*, and *8*. In order to reduce the effect of internal node failure to completeness of the query result, we propagate a query concurrently through a redundant tree topology.

One such method is to execute parallel requests in opposite directions in the overlay. In a recursive-partitioning tree discussed in Section 2.3, a node is responsible for a region in its clockwise-direction; node $n2$ is assigned a region $[n2, n3)$, note that $n2 < n3$. In a counter-clockwise direction tree, a node $n2$ is allocated a region $(n1, n2]$. This approach can compensate missing results in one tree from another if the missing region is not assigned to failed nodes in both trees.

3.2 PonD self-configuration

After acquiring a list of available resources from a resource discovery module, invitations are sent to those nodes to join a PonD of a job submitter. As invited nodes join a PonD, deployed HTC middleware takes roles of task execution and monitoring. In C-PonD, which is an implementation of PonD with Condor, we leverage system configuration "CONDOR_HOST" to create/join a PonD dynamically.

Condor job scheduling is performed at *condor_negotiator* daemon with available resource information and inactive job lists that are fetched at the time of negotiation from the *condor_collector* daemon. Generally, the two daemons run at the central manager node, while worker nodes (*condor_startd* daemon) and job submit nodes (*condor_schedd* daemon) identify the central manager using a condor configuration value "CONDOR_HOST", which can be set independently for daemons on the same resource. The "CONDOR_HOST" configuration can be changed at run-time by using Condor commands (e.g., *condor_config_val* and *condor_reconfig*).

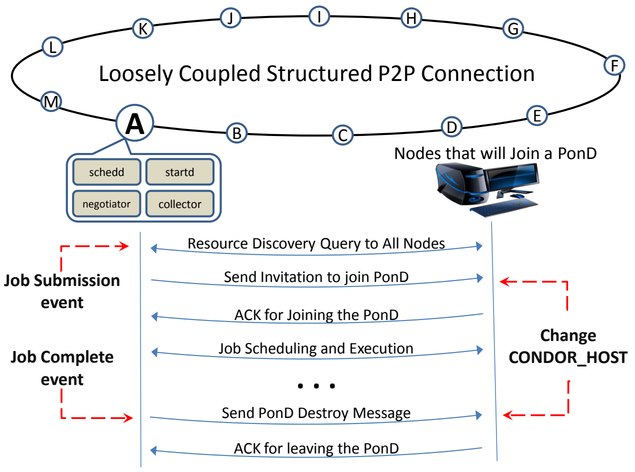

Loosely Coupled Structured P2P Connection
A
B
C
D
E
F
G
H
I
J
K
L
M
schedd
startd
negotiator
collector
Nodes that will Join a PonD
Job Submission event
Resource Discovery Query to All Nodes
Send Invitation to join PonD
ACK for Joining the PonD
Job Complete event
Job Scheduling and Execution
Change CONDOR_HOST
Send PonD Destroy Message
ACK for leaving the PonD

Figure 4: Steps to create a C-PonD through dynamic condor re-configuration.

Figure 4 shows an overall procedure of creating a PonD with Condor for job execution. In the figure, all resources are connected through a structured P2P network, and every node runs *condor_negotiator, condor_collector, condor_startd,* and *condor_schedd* daemon locally. We will show how Node A creates a PonD for a job execution.

- Upon detecting a job submission, Node A sends a resource discovery query with a job requirement, rank criteria, and the number of required nodes, which are derived from the ClassAds of the local submission queue in the *condor_schedd*.
- Upon the query completion, Node A sends a PonD join invitation to nodes that are discovered from the query. When receiving the invitation, nodes that accept it set "CONDOR_HOST" value of *condor_startd* daemon to the address of Node A.
- Nodes send ACK message to confirm joining the PonD.
- When a PonD is created, Node A becomes the central manager of the PonD, and *condor_negotiator* daemon of Node A schedules inactive jobs to available resources. The scheduled jobs are executed using *condor_starter* of worker nodes and *condor_shadow* daemon of Node A.
- Upon job completion, Node A sends a PonD destroy message to nodes in the PonD. When receiving the message, nodes set "CONDOR_HOST" value to the address of itself.
- Each node sends an ACK message to Node A when leaving the PonD.

Note that every node in the resource pool can create a PonD. It is also possible for a node to discover available resources for other PonDs in case a central manager of a PonD is not capable of executing resource discovery query.

3.2.1 Leveraging virtual networking

C-PonD is designed to provide an infrastructure that supports scaling to a large number of geographically distributed resources in a wide area network — a common environment in Desktop Grids. Due to the constraints of Network Address Translation (NAT)/firewalls, however, supporting direct communication between peers for PonD invitation/join, job execution/monitoring using Condor daemons is one of essential features for a successful deployment in the real-world. In order to address this requirement, we adopt IP-over-P2P (IPOP) [26, 27] to build a virtual networking overlay that enables routing IP messages atop a structured P2P network. This approach has benefits with respect to self-management, resilience to failures, and ease-of-use from the user's perspective. The practicality of this approach is well presented by GridAppliance [28] as the authors demonstrated over several years of operation in a WAN environment.

These network challenges present a problem for systems deriving from Condor GlideIn or Flocking, which need publicly available IP addresses on a pilot node, worker/submitter, or central manager node. C-PonD's use of virtual networking allows direct messaging between peers after an IP-layer virtual overlay network has been established.

3.2.2 *Multiple Distinct Jobs Submission*

In the case that multiple jobs with distinct requirements are submitted concurrently by a *condor_schedd*, discovered resources need to be distinguishable by the initiating query. Let us assume that two distinct jobs, $\mathcal{J}_i$ and $\mathcal{J}_k$, are submitted with requirements $\mathcal{R}_i$ and $\mathcal{R}_k$, and a list of discovered resources is $\mathbb{N}_i$ and $\mathbb{N}_k$, respectively. If $\mathcal{R}_i \subset \mathcal{R}_k$ satisfies, jobs in $\mathcal{J}_i$ might suffer from the dearth of resources when jobs in $\mathcal{J}_k$ are scheduled to resources in $\mathbb{N}_i$; $\mathcal{R}_i \subset \mathcal{R}_k$ means that jobs in $\mathcal{J}_k$ can be executed in resources of $\mathbb{N}_i$, but there is no guarantee that jobs in $\mathcal{J}_i$ can be executed in nodes of $\mathbb{N}_k$. In order to deal with this issue, we add a ClassAds attribute dynamically to the *condor_startd* about the ID of a job that matched during resource discovery. Jobs matching the ID will be prioritized to run on that resource.

3.2.3 *Minimum Inter-Scheduling Time*

Equation 2 describes the effect of the minimum inter-scheduling to the expected queueing time of a job before scheduling. Unlike in a typical Condor environment, where one negotiator may deal with multiple job submitters, a negotiator in a C-PonD will handle a single job submitter so that we can decrease the minimum inter-scheduling time significantly without posing large processing overheads to a *condor_negotiator* daemon.

3.3 Discussion

Overheads : While Condor provides a scalable service by deploying multiple managers and fault-tolerant capability with duplicated fail-over servers, the administrative overhead to configure and manage these servers is non-negligible and can incur significant communication costs at scale. In contrast, C-PonD self-configures manager nodes on demand, and because of its ability to reuse unmodified middleware, it can be extended to discover and self-configure nodes for high-availability fail-over services. Though a PonD is still a centralized architecture with a central manager node (a PonD creator) and multiple worker nodes, failures are isolated. A crash of a central manager node does not keep other nodes from creating other PonDs. Worker nodes that are registered at a PonD periodically check the status of the central manager of the PonD. If the central manager is detected to be offline or inactive, it withdraws from the PonD.

Middleware reuse : C-PonD uses unmodified Condor binary files. This is made possible by the use of a decentralized resource discovery module and run-time configuration

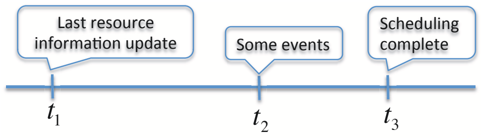


Figure 5: Based on the event at $t2$ (e.g., attribute value change, node crash), scheduling result at $t3$ might be stale.

scripts. By using Condor intact, we reduce the possibility of bugs in scheduling and job execution/monitoring due in large part to the over 20 years use and development of Condor. This feature differentiates C-PonD from many other decentralized HTC middleware which try to build a system from scratch. Reusing the ClassAds module for matchmaking allows us to inherit most of characteristics of ClassAds, such as regular expression matching and dynamic resource ranking mechanisms. No source code modification also implies that our approach can be applied to other HTC schedulers (e.g., XtremWeb [29], Globus toolkit, and PBS).

Resource fair sharing : In order to guarantee fair resource sharing amongst nodes, a manager node in Condor records job priority and user priority. Based on the priority, different users can get different levels of services when there is a resource contention. While there is no central point that keeps global information across all nodes in C-PonD, we can leverage DHT, a distributed shared storage in a P2P, by mapping a job submitter ID to a DHT key.

4. EVALUATION

In this section, we evaluate PonD from different perspectives. We use first-order analytical models to determine resource discovery latency and bandwidth costs. From there, we perform a simulation-based quantitative analysis of C-PonD in terms of scalability of scheduling throughput with respect to the number of resources and the matchmaking result staleness under various attribute dynamics and node uptime. By *staleness*, we check if scheduled resources still satisfy query requirements after matchmaking. Due to discrepancy between the timestamp of resource information and matchmaking, scheduled nodes might no longer satisfy requirements after matchmaking. For instance, in Figure 5, a scheduling is completed at t_3 with resource information that was updated at t_1, where $t_1 < t_3$. If an event happens at $t2$ that changes status of the scheduled nodes, such as attribute value change or crash, the scheduling result that was conducted with resource information at $t1$ might not satisfy job requirements at $t3$, and we define this result discrepancy as *stale*. We complete the evaluation by examining the performance of C-PonD prototype deployed on actual wide-area network testbeds.

4.1 System analysis

Scheduling Latency : The scheduling time of C-PonD with respect to the number of resources (N) is primarily determined by the latency of a resource discovery query. A recursive partitioning tree is known to have $O(\log N)$ tree depth [15], which determines the query latency, as the number of nodes increases. Note that the query latency can be decreased using the *First-Fit* mode described in Section 3.1.3. After discovering available resources, pool invitations are sent in parallel (with $O(\log N)$ routing cost) and a PonD is created as worker nodes join with a cost of $O(1)$. Thus, the asymptotic complexity of the time to create a C-PonD is $O(\log N)$.

Condor performs matchmaking sequentially node by node, which results in linearly increasing pattern with respect to the number of resources. Though the scheduling latency might not be a dominant factor in a pool with modest number of resources considering the overall job execution time (such as within a C-PonD), the linearly increasing pattern can become a non-negligible overhead as the pool size increases.

If we consider Condor-flocking in a large scale environment, the matchmaking time at each central manager might not be a prevailing factor for a scheduling latency, because we can assume that the number of resources will be evenly distributed amongst flocking servers. However, as Equation 2 implies, the scheduling time can be affected by the minimum inter-scheduling time while traversing different central managers sequentially. Thus, we can expect that the scheduling time increases with $O(S_f)$, where S_f means the number of flocking servers to traverse.

Other than the number of resources, the scheduling latency is dependent on the number of jobs for matchmaking. Condor has *auto-clustering* feature to improve job scheduling throughput. At the time of scheduling, *condor_schedd* daemon checks *SIGNIFICANT_ATTRIBUTES* (SA), which is a subset of attributes of *condor_startd* and *condor_schedd*. Job ClassAds whose values of SA are same are grouped in a cluster, and they are scheduled at once, which improves the scheduling throughput [30]. The auto-clustering can be also applied to C-PonD to decrease the number of resource discovery queries.

Network Bandwidth Overheads : C-PonD incurs bandwidth consumption during resource discovery query processing, PonD join invitation messages, and *condor_startd* ClassAds updates while a PonD is active. In order to understand query overheads, we assume that the number of resources is N, the size of default arguments (e.g., task name, query range) is S_D, a size of query requirement is S_Q, a P2P address size is S_A, and the number of required resources is k. The bandwidth consumption between a child and parent node is $(S_D + S_Q + S_A \times k)$ for a query distribution and a result aggregation. Using a recursive-partitioning tree, a message is routed only to 1-hop neighbors, so the total query bandwidth usage is $(N-1) \times (S_D + S_Q + S_A \times k)$. In order to analyze per edge bandwidth consumption, dividing total bandwidth consumption by the number of edges gives constant factors $(S_D + S_Q + S_A * k)$. In other words, (N-1) edges are traversed in a query, and each edge corresponds to two messages (send/receive), thus the average per-edge bandwidth consumption follows $O(S_D + S_Q + S_A \times k)$. In the typical case where k is a constant, the average per-edge bandwidth consumption is $O(1)$. In a tree, nodes located closer to the root node are likely to have more child nodes. Based on the underlying P2P-topology of C-PonD, the number of neighboring connections of a node is bounded by the log of total number of nodes. Accordingly, in the worst case, a node is responsible for O($\log N$) edges.

Unlike C-PonD, Condor incurs inactive job fetching overheads during the negotiation phase and periodic *condor_startd* ClassAds update regardless of job submission status. Assuming that a pool size is N, S_c is the size of *condor_startd* ClassAds (usually several KBytes in a typical Condor setting), and T_u is the update period (300 seconds in default setting), the update bandwidth is $N \times S_c \times \frac{1}{T_u}$ per second at the central manager node. As T_u gets longer, the overhead of the central manager decreases, but the longer update period can result in a stale scheduling result in case dynamic attributes are specified as requirements.

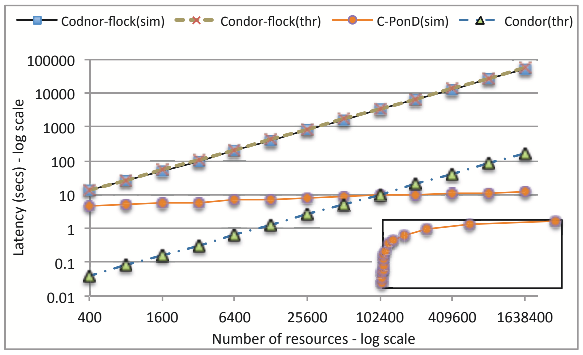


Figure 6: Scheduling latency increasing pattern of Condor-flock, C-PonD, and Condor in various pool sizes. *sim* shows a simulation result, and *thr* shows a theoretical analysis. A bottom-right inner-graph shows a logarithmic pattern of C-PonD resource discovery time.

Matchmaking Result Staleness : Condor relies on periodically updated resource information for matchmaking, therefore the scheduling result correctness depends on the dynamics of attributes and the information update period. In addition, the varying uptime of worker nodes can also influence the result correctness. Condor adopts soft-state to keep *condor_startd* ClassAds of available resources, and a resource that has not updated its ClassAds during the past 15 minutes (default configuration) are removed from the worker node list, which can also result in stale matchmaking results during the soft-state time span.

In C-PonD, the matchmaking is performed using local ClassAds, so the only component that influences result staleness is the query result transmission time. We compare the matchmaking result staleness of Condor and C-PonD through simulations under a controlled environment with synthetic attribute dynamics and resource uptime.

4.2 Simulations

In order to evaluate C-PonD on a large scale simulated environment, we implemented an event-driven simulation framework using C++ language and Standard Template Library (STL) aimed at minimizing the memory footprint in order to perform a simulation of millions of nodes[2]. We used Archer [31] in order to run simulations on distributed computing resources efficiently. After a resource pool is formed with a given number of resources, nodes remain in the pool until a simulation finishes.

4.2.1 *Evaluation on Scheduling Scalability*

Figure 6 presents the scheduling latency of Condor and Condor with flocking, and resource discovery time of C-PonD in the Y axis and the number of resources in a pool in the X axis (both represented in log-scale). The dotted lines show expected latency based on the analysis, and the solid lines show the response time from simulations. Because

[2]https://github.com/kyungyonglee/PondSim

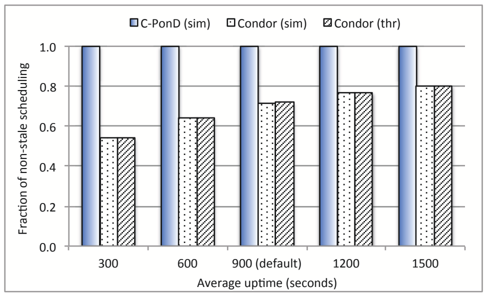
there currently is no enviromnent capable of precise simulation of Condor at the scale of millions of nodes, and because the latency of each method is dependent on constants associated with system environments (e.g., network latency, computing capacity of scheduler, number of resources in each pool for flocking, available bandwidth), the focus of this analysis is to highlight trends of increasing scheduling latency as the number of resources increases and gauge the expected magnitude of a system where the latency of decentralized discovery becomes apparent.

In the simulation, the expected scheduling time of Condor is derived from experiment results from Bradley et. al [32] — the latency of one round of matchmaking, which schedules a single job (or multiple jobs clustered by *auto-clustering*) to top-ranked resources, being one second in a pool of 10,000 worker nodes. Based on the fact that Condor performs sequential traversing of all worker nodes, we apply a linear model to predict the scheduling time for different numbers of worker nodes. For C-PonD, we assume a widely distributed WAN resource pool and set the transmission delay of an edge between 50 ms and 300 ms following a uniform distribution [33]. In Condor-flocking, we set an average number of resources in a pool as 100 with default negotiation cycle of 60 seconds and the minimum inter-negotiation cycle of 20 seconds, which is the default Condor configuration value. In addition to the simulation, we also present a analytical result based on Equation 2.

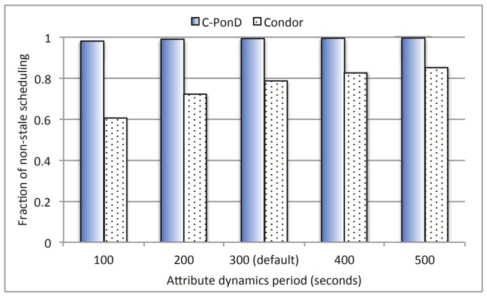
As shown in the figure, Condor and Condor-flock have a linearly increasing pattern, and C-PonD shows a logarithmic pattern as the number of resources increases; this can be clearly observed in the bottom-right inner graph, where the horizontal and vertical axis values are represented in the absolute value scale. The figure also shows that results from Equation 2 the almost identical values of Condor-flock simulation and theory. Though the magnitude of latency can vary according to system configurations, we can observe that the latency of C-PonD outperforms Condor when the number of resources is larger than 100,000, which asserts the necessity of careful system design with respect to the target number of resources. From this simulation, in summary, we would like to address that the possibility of scheduling throughput limitations of centralized HTC middleware in a large scale can be relieved by leveraging a decentralized resource discovery mechanism, while supporting existing HTC middleware for job scheduling.

4.2.2 Matchmaking Result Staleness

In this section, we discuss the staleness of a scheduling result of C-PonD and compare it to Condor to illustrate the ability of C-PonD to cope with the dynamics of resource uptime and attributes. Condor relies on soft-state to determine if a worker node is alive or not; at a default system configuration, a worker node that does not update ClassAds for the last 15 minutes is deemed to be non-accessible. Thus, scheduling to a non-accessible worker node can be made within the time-span of soft-state period. In contrast, in C-PonD, heart-beat messages are exchanged between resources every few seconds, allowing the underlying P2P network to deal with node failures in a decentralized fashion. In order to address the staleness of a scheduling result quantitatively in a controlled environment, we perform a simulation by making worker nodes join and leave a pool dynamically with different mean session times following exponential distribution. Upon joining a pool, a node updates its ClassAds to the central manager every 300 seconds until it leaves the pool. The central manager keeps the ClassAds until the soft-state period expires.



(a) Scheduling based on non-stale information for simulated Condor and C-PonD, and based on Equation 4 (Condor-thr)



(b) The scheduling result of Condor varies according to the dynamics of attributes

Figure 7: Scheduling result staleness of C-PonD and Condor

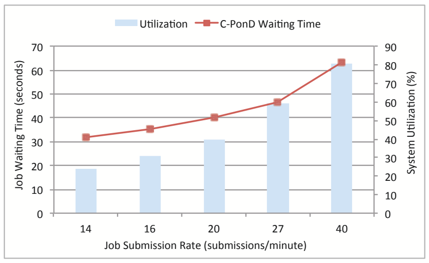
Let us assume that the default ClassAds update period of Condor as T_d, a soft-state expiration time as T_s. Given an exponentially distributed join/leave event of worker nodes with a mean event rate of λ and assuming that a worker node is alive at time t_0, the probability of the node being inactive - the next event happens - after t elapses is defined as $\lambda e^{-\lambda t}$. If $0 < t \leq T_s - T_d$, given that $T_s > T_d$, the inactive node is deemed as alive to the central manager. If $T_s - T_d < t < T_s$, the node is deemed to be down or up according to the last ClassAds update time. Given the memoryless property of exponential distribution, we can calculate the probability of a ClassAds of inactive node still exist at a central manager as $\frac{T_s - t}{T_d}$. Finally, given that a node is alive at t_0, we can calculate the probability of a scheduling result being stale after t elapses, $P(Stl_t)$, as

$$P(Stl_t) = \int_0^{T_s - T_d} \lambda e^{-\lambda t}\, dt + \int_{T_s - T_d}^{T_s} \lambda e^{-\lambda t} \times \frac{T_s - t}{T_d}\, dt \tag{3}$$

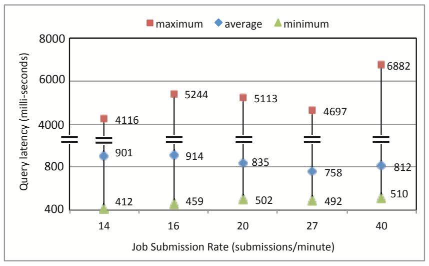
Due to the alternating event of join/leave, the probability of a node being active at arbitrary time t_0 is $\frac{1}{2}$. Thus we can calculate the probability of scheduling result being non-stale after t elapses from t_0 as

$$P(non - stl_t) = 1 - \frac{1}{2} \times P(Stl_t) \tag{4}$$

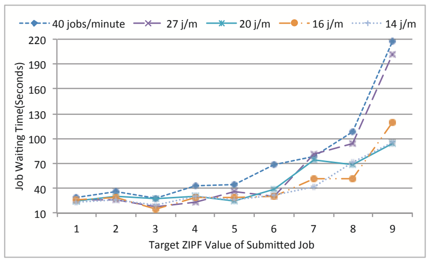
In order to validate Equation 4 and to present a quantitative comparison between C-PonD and Condor, we per-



(a) Job waiting time and utilization



(b) Resource discovery latency



(c) Job wait time based on ZIPF values

Figure 8: The performance metrics measured from a deployment of C-PonD on FutureGrid

formed a simulation in a pool of 10^6 worker nodes setting edge latencies between 50 and 300 msec uniformly with the different value of worker node's join/leave event rate λ – $\frac{1}{300}, \frac{1}{600}, \frac{1}{900}, \frac{1}{1200}, \frac{1}{1500}$ per second. For a Condor pool, we adopted the default system configuration value. For C-PonD, we assume the node join and departure are handled via the underlying P2P network topology. Figure 7a shows the fraction of non-stale scheduling results according to the different average uptime. The left-most solid-bar shows the simulated value of C-PonD, the dotted grid bar shows the simulation result of Condor, and the diagonal bar shows an expected value calculated based on Equation 4 for Condor. We can observe that C-PonD has almost no impact for the given worker nodes uptime, because the only factor that can influence the result staleness is the aggregated result propagation time that is of the order of tens of seconds in a 10^6 pool. In case of Condor, the staleness of the scheduling result has a non-negligible impact based on the dynamics of worker nodes. For instance, 20% of scheduling results are stale when the average uptime of a node is 25 minutes. The results also claim the correctness of the analytical model of the scheduling result non-staleness in Equation 4 - the similar value of Condor(sim) and Condor(thr).

Next, we present the effect of attribute value dynamics to the scheduling result. In C-PonD, the matchmaking is performed with resource information that is available from the local *condor_startd* daemon. Otherwise, in Condor, the matchmaking result might be as stale as the information update period (300 seconds in the default Condor configuration). In order to model scheduling with a requirement of non-uniform likelihoods of resource matching, we add a synthetic ClassAd attribute, *ZipfAttribute* whose value follows Zipf-distribution. A worker node selects an integer value from one to ten with skew value of 1.0 and heavy portion of the distribution at one (35% of nodes). At the simulation, a node changes the *ZipfAttribute* value at a rate of λ following the exponential distribution. The scheduling staleness is measured as the fraction of nodes that still satisfies the requirement after the matchmaking. We do not show detailed analysis for this model due to the similarity with Equation 3.

The vertical axis of Figure 7b shows the fraction of non-stale scheduling result, and the horizontal axis shows the mean time of attribute change event. As shown in the figure, Condor has a higher impact than C-PonD as an attribute value changes more dynamically. In case of Condor, about 20% of scheduling results are stale when the average attribute value change rate is the same as Condor ClassAds update period (300 seconds).

4.3 Deployment on FutureGrid

In order to demonstrate the correctness of the implementation and the feasibility of C-PonD approach as a real-world HTC middleware in a wide area network environment, we conducted experiments on a cloud computing infrastructure, FutureGrid [6]. A total of 240 Virtual Machine (VM) instances were deployed across USA using a VM image that has C-PonD modules installed. Specifically, we setup 80 VMs at the University of Chicago, 80 VMs at Texas Advanced Computing Center (TACC), 70 VMs at San Diego Supercomputer Center (SDSC), and 10 VMs at the University of Florida. Each VM instance was assigned a single-core CPU and 1 GB of RAM.

In order to evaluate the system in a realistic setup created a synthetic job submission scenario by referencing DAS-2 trace from the grid workload archive [34]. From the trace, we extracted memory usage information, the number of independent concurrent execution for each job, and the running time of each job. An extra resource attribute named *ZipfAttribute* was added to resource ClassAds to allow the experiments to vary the likelihood of resource discovery queries to find available resources. During the experiment, each submitted job selects an integer target value from one to ten following a Zipf-distribution as a requirement. Each worker node also selects *ZipfAttribute* value at the C-PonD initialization time.

During the match-making process, nodes that satisfy requirements extracted from the DAS-2 trace file and whose *ZipfAttribute* is same as the selected Zipf value are returned. In a real grid computing scenario, finding nodes with high ZipfAttribute value can be thought of as searching for best machines with higher capabilities than other machines in a resource pool. We differentiated average job inter-arrival time that follows the exponential distribution to observe the system performance under different system utilizations. The experiment is conducted for two hours per each job submission rate. A job that is composed of multiple independent concurrent tasks is submitted at an arbitrary node while keeping the overall job submission rate of the system.

Figure 8a shows the average job waiting time of C-PonD at the primary vertical axis under different job submission rates shown in the horizontal axis. The job waiting time is the elapsed time since the job submission until the job execution begin time. The secondary vertical axis shows system utilization that is calculated as

$$\frac{\sum JobRunningTime}{ExperimentTime * NumberofResources}$$

As the job submission rate increases, the system utilization and job waiting time also increase due to resource contention. In order to clarify the reason for longer waiting time as the job submission rate increases, we present the average, minimum, and maximum query latency for different job submission rates in Figure 8b. In Figure 8b, we can observe that the resource discovery query latency does not have a strong correlation with the job submission rate; regardless of the contention, it took less than a second to traverse 240 resources distributed in a WAN network. We can also observe the occasional longer query latency that might be caused by internal lagging nodes in a tree during the aggregation task, which can be improved by heuristics dealing with abnormalities in a tree, such as First-Fit and redundant topology in Section 3.1.3 and 3.1.4 (which were not applied in this experiment).

Figure 8c shows an average job waiting time of C-PonD based on the different target *ZipfAttribute* value under different job submission rates. The horizontal axis shows the target Zipf value at a submission, and the vertical axis shows the corresponding wait time. As we can see, the waiting time gets longer when the number of requirement satisfying resources is reduced (e.g., higher *ZipfAttribute* value). It can be explained as follow: a job is composed of a large number of independent concurrent tasks that can run on different machines. Thus, when a job with requirements of high *ZipfAttribute* is submitted, a resource contention is likely to happen among multiple independent tasks in a job.

These experiments validates the functionality of C-PonD in a real-world WAN environment while showing a good query response time regardless of the job submission rate and resource utilization.

4.4 C-PonD running on PlanetLab

In this experiment, we observe the dynamic behavior of C-PonD creation through resource status snapshots over a 65 hour period while running C-PonD on 440 PlanetLab nodes. After installing C-PonD module on each PlanetLab node, we add the geographic coordinate information that is extracted from the IP address of each node as *condor_startd* ClassAds attributes. We leverage this attribute to build a query requirement. In a HTC pool that is deployed on a widely distributed environment, we can explicitly specify a region of job execution using a geospatial-aware query based on latitude/longitude range. This is a useful scenario in case a communication cost is a substantial factor determining job performance. The flexibility of adding new attributes addresses an advantage of C-PonD over other decentralized resource discovery systems.

In Figure 9, *Req.1* region in the horizontal axis, jobs are submitted with a requirement that says *ZipfAttribute* is one of the values from one to ten. In the *Req.2* region, jobs are submitted with a requirement of worker nodes being in America. In *Req.3* region, jobs are submitted with a requirement of worker nodes being in Europe, and *Req.4* requires worker nodes being in Asia. The vertical axis shows the number of available resources in America (top horizontal line), Europe (middle line), and Asia (bottom line) continent. The number of running jobs at the time is shown at

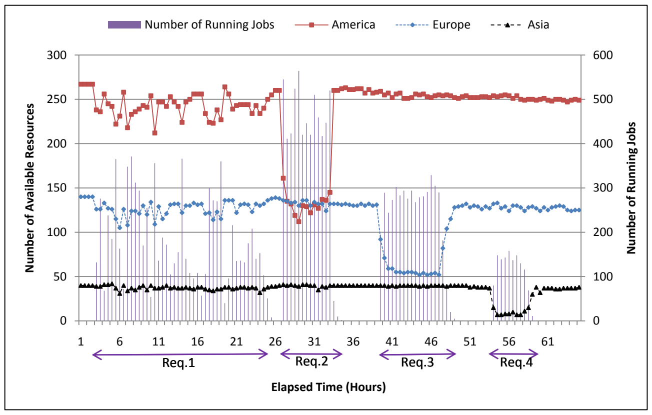


Figure 9: The number of available resources and running jobs status snapshots of C-PonD execution on 440 PlanetLab nodes during 65 hours of experiment. In Req. 2, 3, and 4 region, jobs runs on America, Europe, and Asia nodes, respectively.

the vertical bar whose value can be referenced at the secondary vertical axis.

We controlled the job submission rate not to overwhelm PlanetLab nodes, as they are shared resources not dedicated to a single user. Based on observation with varying requirements, C-PonD showed a flawless operation for a PonD creation and job execution.

From the figure, we can observe that the number of running jobs at a time is more than the number of claimed machines (equal to total number of nodes minus the number of available nodes). The reason is as follows: if a worker node has multi-core CPU, the *condor_startd* daemon recognizes each core as a distinct worker node, and those distinct worker nodes execute jobs independently. Thus, a node with multiple cores can run multiple jobs at a time.

5. RELATED WORK

Decentralized resource discovery methods are proposed from various aspects. In a structured P2P network, Sword [19] and Mercury [21] enhance DHTs to support range queries by mapping attribute names and values to the DHT key. Kim et. al [20], Squid [22], and Artur et. al [35] discuss resource locating methods on a multi-dimensional P2P network. Kim et. al [20] maps a resource attribute to one dimension in CAN [36], and a requirement confirming zone is created from a query requirement. Squid [22] leverages Space Filling Curve (SFC) to convert a multi-dimensional space to a one-dimensional ring space, where the locality preserving feature of SFC allows a range query. However, the limitation of locality-preservation for dimensions over five limits scalability of Squid. Artur et. al [35] converts multi-dimensional spaces into one dimension ring space. The correlation between multiple dimensions and attribute values are leveraged for match-making. This algorithm has a scalability limitation for a large number of attributes. Similar to our work, Armada [18] use a tree-structure for match-making by assigning an Object ID based on an attribute value, where a partition tree is constructed based on the proximity of the object ID. The resource discovery module of PonD has advantages in the aspects of rich query processing capacity and appropriateness for dynamic attributes over these works.

In an unstructured network, Iamnitchi et. al [37] proposes a flooding-based query distribution for match-making on a P2P network with query forwarding optimization method leveraging previous query result and similarity of queries. Due to the characteristics of flooding, duplicated messages hurt the query efficiency. Shun et. al [38] uses super-peers which keep the resource status of worker nodes. In order to minimize information update overhead, they use threshold to decide whether to distribute updated resource information. Thomas et. al [39] uses hybrid approach of flooding-based approach and structured id based propagation method. Different from the resource discovery module of PonD, these methods do not guarantee that a resource discovery query can be resolved even there is a node that satisfies job requirements. Periodic resource information update can also result in result-staleness.

MyGrid [40] provides a software abstraction to allow a user to run tasks on resources that are available to the user regardless of scheduler middleware. Based on MyGrid work, OurGrid [41] toolkit provides a mechanism to allow users to gain access to computing resources on other administrative domains while guaranteeing a fair resource sharing among resources that are connected through P2P network.

BonjourGrid [42] and PastryGrid [23] are decentralized grid computing middlewares. For resource discovery, BonjourGrid [42] uses publish/subscribe multicast mechanism. Without structured multicast mechanism, resource discovery results can flood the query initiating node. Due to reliance on IP-layer multicast, it has no guarantee to work on WAN environment. We solved this issue by leveraging P2P-based virtual overlay network. PastryGrid [23] is built on a structured P2P network, Pastry [43], and the resource discovery is performed by sequentially traversing nodes until a required number of nodes is discovered. Though they did not measure efficiency of resource discovery method, the cost of sequential traversing (e.g., $O(N)$, where N is the number of query satisfying nodes) is much more expensive than our parallel resource discovery mechanism, which is $O(logN)$.

MyCluster [44] presents a method to submit and run jobs across different administrative cluster domains. In their approach, a proxy is used to reserve a specific amount of CPUs in other clusters for some periods of time. The proxy is also responsible for setting task execution environment across different clusters in a user-transparent way. Due to the resource reservation mechanism for a specific amount of time, resource underutilization can happen after job completion.

Celaya et. al. [33] proposes a decentralized scheduler for HTC using a statically built tree for resource information aggregation and resource discovery. A parent node keeps aggregated resource information of a sub-tree rooted at itself, and the information is leveraged for scheduling. Due to the periodic resource information update, the scheduling result might be misleading, and it supports a limited number of attributes for resource discovery (e.g., memory, disk space). Without a solid mechanism to handle internal node failures in a statically built tree, the system can not guarantee a reliable service.

6. CONCLUSIONS

This paper presents and evaluates PonD, a novel approach for scalable, self-organizing and fault-tolerant HTC service. The system combines a P2P overlay for resource discovery across a loosely-coupled resource "sea" in order to create a small "pond" of resources using unmodified HTC middleware for job execution and monitoring. Our approach removes the need for a central server or set of servers which monitor the state of all resources in an entire pool. A decentralized resource discovery module provides a mechanism to discover a list of nodes in the pool which satisfy job requirements by leveraging a self-organizing tree for a query distribution and result aggregation. Through the first-order performance analysis and simulations, we compared our approach with an existing HTC approach, Condor. In terms of scalability, a job execution pool creation time of PonD grew logarithmically as the number of resources increases. The evaluation on scheduling result correctness under a dynamic environment presents the robustness of our approach to dynamic attribute value changes and worker node churn. In order to demonstrate the validity of our proposed approach, we deployed and experimented a prototype implementation of C-PonD using unmodified Condor binary files leveraging run-time configurations setup in the real-world - PlanetLab and FutureGrid. Although C-PonD implementation is centered on a Condor, the clean separation of decentralized resource discovery and PonD creation module from the HTC middleware makes this approach generalizable and applicable to other HTC middleware, and a further differentiation from other decentralized HTC platforms that try to cover all phases of operation (e.g., resource discovery, scheduling, job execution and monitoring).

7. ACKNOWLEDGMENTS

We would like to thank our shepherds Douglas Thain and Thilo Kielmann, anonymous reviewers, and Alain Roy for their insightful comments and feedback. This work is sponsored by the National Science Foundation (NSF) awards 0751112 and 0910812.

8. REFERENCES

[1] Jim Basney and Miron Livny. *Deploying a High Throughput Computing Cluster*. Prentice Hall, 1999.

[2] David P. Anderson. Boinc: A system for public-resource computing and storage. In *Fifth IEEE/ACM Inter. Workshop on GRID*, 2004.

[3] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, and M. Wilde. Falkon: a fast and light-weight task execution framework. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, 2007.

[4] O. Sonmez, N. Yigitbasi, S. Abrishami, A. Iosup, and D. Epema. Performance analysis of dynamic workflow scheduling in multicluster grids. HPDC '10, 2010.

[5] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. Planetlab: an overlay testbed for broad-coverage services. *SIGCOMM Comput. Commun. Rev.*, 2003.

[6] Gregor von Laszewski and et. al. Design of the futuregrid experiment management framework. In *GCE2010 at SC10*, New Orleans, 11/2011 In Press.

[7] Solomon. M. Ruman. R, Livny. M. Matchmaking: distributed resource management for high throughputcomputing. In *7th HPDC*, 1998.

[8] D.H.J. Epema, M. Livny, R. van Dantzig, X. Evers, and J. Pruyne. A worldwide flock of Condors: Load

sharing among workstation clusters. *Future Generation Computer Systems*, 12:53–65, 1996.

[9] J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke. Condor-g: a computation management agent for multi-institutional grids. In *HDPC*, 2001.

[10] Ali R. Butt, Rongmei Zhang, and Y. Charlie Hu. A self-organizing flock of condors. *J. Parallel Distrib. Comput.*, 66:145–161, January 2006.

[11] I. Sfiligoi. glideinWMS a generic pilot-based workload management system. *Journal of Physics Conference Series*, 119(6):062044–+, July 2008.

[12] P.Oscar Boykin and et al. A symphony conducted by brunet, 2007.

[13] J. M. Kleinberg. Navigation in a small world. *Nature*, 406, August 2000.

[14] V. Vishnevsky, A. Safonov, M. Yakimov, E. Shim, and A. D. Gelman. Scalable blind search and broadcasting over distributed hash tables. *Comput. Commun.*, 2008.

[15] T. Choi and P. O. Boykin. Deetoo: Scalable unstructured search built on a structured overlay. In *7th Workshop on Hot Topics in P2P Systems*, 2010.

[16] A. I. T. Rowstron, A. Kermarrec, M. Castro, and P. Druschel. Scribe: The design of a large-scale event notification infrastructure. In *Proceedings of the Third International COST264 Workshop on Networked Group Communication*, 2001.

[17] J. Kim, B. Bhattacharjee, P. J. Keleher, and A. Sussman. Matching jobs to resources in distributed desktop grid environments. 2006.

[18] D. Li, J. Cao, X. Lu, and K. C. C. Chen. Efficient range query processing in peer-to-peer systems. *IEEE Trans. on Knowl. and Data Eng.*, 2009.

[19] J. Albrecht, D. Oppenheimer, A. Vahdat, and D. A. Patterson. Design and implementation trade-offs for wide-area resource discovery. *ACM Trans. Internet Technol.*, 2008.

[20] Jik-Soo Kim, Peter Keleher, Michael Marsh, Bobby Bhattacharjee, and Alan Sussman. Using content-addressable networks for load balancing in desktop grids. In *HPDC*, 2007.

[21] A. R. Bharambe, M. Agrawal, and S. Seshan. Mercury: supporting scalable multi-attribute range queries. *SIGCOMM Comput. Commun. Rev.*, 2004.

[22] C. Schmidt and M. Parashar. Squid: Enabling search in dht-based systems. *J. Par. Distrib. Comput.*, 2008.

[23] Heithem A., Christophe C., and Mohamed J. A decentralized and fault-tolerant desktop grid system for distributed applications. In *Concurrency and Computation: Practice and Experience*, 2010.

[24] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 2008.

[25] K. Lee and et. al. Parallel processing framework on a p2p system using map and reduce primitives. In *8th Workshop on Hot Topics in P2P Systems*, 2011.

[26] A. Ganguly, A. Agrawal, P. Boykin, and R. Figueiredo. Ip over p2p: enabling self-configuring virtual ip networks for grid computing. In *IPDPS*, 2006.

[27] David Isaac Wolinsky and et. al. On the design of scalable, self-configuring virtual networks. In *Proceed. of Super Computing*, SC '09, 2009.

[28] David Isaac Wolinsky and Renato Figueiredo. Experiences with self-organizing, decentralized grids using the grid appliance. HPDC '11. ACM, 2011.

[29] G. Fedak, C. Germain, V. Neri, and F. Cappello. Xtremweb: A generic global computing system. CCGRID '01, 2001.

[30] condor auto clustering. `https://condor-wiki.cs.wisc.edu/index.cgi/wiki?p=AutoclustingAndSignificantAttributes`.

[31] Renato J. Figueiredo and et al. Archer: A community distributed computing infrastructure for computer architecture research and education. In *Collaborative Computing*, 2009.

[32] D Bradley, T St Clair, M Farrellee, Z Guo, M Livny, I Sfiligoi, and T Tannenbaum. An update on the scalability limits of the condor batch system. *Journal of Physics: Conference Series*, 331(6):062002, 2011.

[33] J. Celaya and U. Arronategui. A highly scalable decentralized scheduler of tasks with deadlines. In *Grid Computing (GRID), 2011*, 2011.

[34] A. Iosup, H. Li, M. Jan, S. Anoep, C. Dumitrescu, L. Wolters, and D. H. J. Epema. The grid workloads archive. *Future Gener. Comput. Syst.*, 2008.

[35] A. Andrzejak and Z. Xu. Scalable, efficient range queries for grid information services. In *P2P '02*, 2002.

[36] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *SIGCOMM*, 2001.

[37] A. Iamnitchi and I. T. Foster. On fully decentralized resource discovery in grid environments. GRID, 2001.

[38] Shun K. K. and Jogesh K. M. Resource discovery and scheduling in unstructured peer-to-peer desktop grids. *Parallel Processing Workshops*, 2010.

[39] T. Fischer, S. Fudeus, and P. Merz. A middleware for job distribution in peer-to-peer networks. In *Applied Parallel Computing. State of the Art in Scientific Computing.* 2007.

[40] Lauro B. C., Loreno F., Eliane A., Gustavo M., Roberta C., Walfredo C., and Daniel F. Mygrid: A complete solution for running bag-of-tasks applications. In *In Proc. of the SBRC*, 2004.

[41] N. Andrade, W. Cirne, F. Brasileiro, and P. Roisenberg. Ourgrid: An approach to easily assemble grids with equitable resource sharing. In *Proceedings of the 9th Workshop on Job Scheduling Strategies for Parallel Processing*, 2003.

[42] Heithem A., Christophe C., and Mohamed J. Bonjourgrid: Orchestration of multi-instances of grid middlewares on institutional desktop grids. *Parallel and Distributed Processing Symposium*, 2009.

[43] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems, 2001.

[44] E. Walker, J.P. Gardner, V. Litvin, and E.L. Turner. Creating personal adaptive clusters for managing scientific jobs in a distributed computing environment. In *Challenges of Large Applications in Distributed Environments*, 2006.