ARTICLE IN PRESS

Available online at www.sciencedirect.com

SCIENCE DIRECT®

J. Parallel Distrib. Comput. ▮▮▮ (▮▮▮▮) ▮▮▮–▮▮▮

Journal of Parallel and Distributed Computing

www.elsevier.com/locate/jpdc

A self-organizing flock of Condors

Ali R. Butt*, Rongmei Zhang, Y. Charlie Hu

School of Electrical and Computer Engineering, Purdue University, West Lafayette, IN 47907, USA

Received 4 December 2003; received in revised form 27 June 2005; accepted 27 June 2005

Abstract

Condor enables high throughput computing using off-the-shelf cost-effective components. It also supports flocking, a mechanism for sharing resources among Condor pools. Since Condor pools distributed over a wide area can have dynamically changing availability and sharing preferences, the current flocking mechanism based on static configurations can limit the potential of sharing resources across Condor pools. This paper presents a technique for resource discovery in distributed Condor pools using peer-to-peer mechanisms that are self-organizing, fault-tolerant, scalable, and locality-aware. Locality-awareness guarantees that applications are not shipped across long distances when nearby resources are available. Measurements using a synthetic job trace show that self-organized flocking reduces the maximum job wait time in queue for a heavily loaded pool by a factor of 10 compared to without flocking. Simulations of 1000 Condor pools are also presented and the results confirm that our technique discovers and utilizes nearby resources in the physical network.
© 2005 Elsevier Inc. All rights reserved.

Keywords: Distributed resource discovery; Condor; Flocking; Peer-to-peer

1. Introduction

The complexity of today's scientific applications and their exponentially growing data sets necessitate utilizing all available resources. The economic constraints of deploying specialized hardware entail leveraging off-the-shelf equipment to satisfy the growing need for computing power. Sharing of these resources poses design challenges especially in resource management and discovery [3]. The computational grid [16], popularized by systems such as Globus [21,15] and Condor [29], provides ways for applications to be spread across multiple administrative domains. Issues of access control, resource management, job scheduling, and user management are addressed at great length in these systems. On the other hand, the peer-to-peer (p2p) overlay networks, motivated by file-sharing systems such as Napster [34], Gnutella [18], and Kazaa [45], and formalized by systems such as CAN [40], Chord [47], Pastry [44], and Tapestry [52], have demonstrated the ability to serve as a robust, fault-tolerant, and scalable substrate for a variety of applications. Examples of p2p applications include distributed storage facilities [43,10], application-level multicast [6,54,17], and routing in mobile ad hoc networks [24]. In this work, we develop a new p2p application that utilizes p2p techniques to facilitate the discovery of remote resources. Although the results that will be presented are achieved via an innovative marriage of the flocking facility [11] in Condor and the proximity-aware p2p routing substrate offered by Pastry [44,5], the scheme is applicable to other platforms.

Condor [29] enables high throughput computing using off-the-shelf cost-effective components. It can be employed to manage dedicated resources such as rack clusters as well as to harness the idle cycles on desktop machines. Condor also supports application checkpointing by providing libraries that can be linked with the application. A Condor pool is statically configured to use a selected machine as the central manager. The task of the manager is to schedule jobs to various idle resources in the pool, and to facilitate the migration of running jobs. Under normal conditions,

* Corresponding author.
E-mail addresses: butta@purdue.edu (A.R. Butt), rongmei@purdue.edu (R. Zhang), ychu@purdue.edu (Y.C. Hu).

0743-7315/$ - see front matter © 2005 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2005.06.022

a job waits in a queue until the central manager can find an appropriate resource in the pool to run it on.

There are two constraints that can limit the potential of Condor to share available resources. First, the size of individual pools is limited by the resources available to an organization. There is an ever growing need to collaborate and share resources with other organizations to support higher throughput. Second, the failure of the central manager can result in inability to process new jobs. For these reasons, it is desirable to develop an automatic mechanism that minimizes the downtime and provides continuity of service for new jobs.

Condor supports a mechanism referred to as flocking [11] that provides a way for sharing resources among multiple pools. This mechanism is static and requires manual configuration. In the dynamic situations of real world where the availabilities and sharing preferences of individual pools vary, a self-organizing, scalable, and robust mechanism is needed to fully exploit the potential of resource sharing across multiple administrative domains.

The p2p overlay networks can help in automating the discovery of appropriate resource pools across administrative domains. Although other means of discovering resources such as those in Globus [15] can also be used, p2p systems have the added advantage that they are robust, scalable, and relatively simple to deploy. The p2p overlays are ideal for situations where nodes often come and leave, justifying our choice of using them in this scenario. Moreover, Pastry—our choice of p2p overlay—is locality-aware, implying that a resource location service based on it can locate a resource close to the requesting node among all available resources in the physical network. This proximity leads to saved bandwidth in data transfer and faster job issue and completion. Besides resource discovery, p2p overlays provide fault tolerance that can be leveraged to provide automatic central manager replacement within a pool.

The main contributions of this work are as follows:

- We describe a p2p-based flocking scheme that allows distributed Condor pools to self-organize into a p2p overlay and locate nearby resource pools in the physical network for flocking. Our approach provides a scalable and flexible method of utilizing the flocking mechanism supported by Condor by increasing the opportunity to flock via dynamic discovery of remote resources.
- We present a prototype implementation of the proposed scheme which shows that the scheme can be easily incorporated into an existing platform.
- We evaluate the proposed scheme via measurements on our prototype implementation running on several small Condor pools distributed across the United States and Europe, as well as through large-scale simulations involving 1000 Condor pools. In this work, we leverage PlanetLab [36] resources for more thorough experiments than those reported in our earlier work [4].

The rest of the paper is organized as follows. Section 2 gives an overview of Condor and structured p2p overlay networks, which serve as building blocks for our approach. Section 3 presents our proposed p2p scheme for creating a self-organized flock of Condor pools. Section 4 presents the architecture of the developed software. Section 5 presents an evaluation and analysis of the proposed scheme. Section 6 presents a survey of work related to our proposed scheme. Finally, Section 7 provides concluding remarks.

2. Enabling technologies

Since the proposed work leverages the idle-cycle sharing facilities of Condor [29] and Pastry [44], a scalable, self-reorganizing, p2p routing and object location infrastructure, we first give a brief background on Condor, flocking in Condor, and Pastry in this section.

2.1. Condor

Condor provides a way for users to solve scientific problems using the resources available to them rather than expensive special-purpose hardware. A Condor pool is created by running the Condor software on all the resources where compute cycles are available, for instance instructional laboratory machines in an academic setting. The software monitors the state of each resource and determines whether it is idle or not. If the resource is idle, Condor can harness its computing power by running computations on it. In this way, resources that would otherwise be unused form a computing cluster. Each pool has a central manager—a machine in the pool chosen for collecting job requests and scheduling jobs to run on the idle machines in the pool. When a user submits a job, it is placed on a queue on the submission machine. The job waits in the queue until an appropriate resource is found on which the job can be run. To facilitate the search for a matching resource for the job, Condor uses an extensive resource description language ClassAds [37] that allows for specification of resources and the jobs they can support, as well the specification of jobs and the nature of resources they require to run. Using this language, the submission machine provides a description of the job to the central manager, which finds a match using a matchmaking technique [39]. In addition, Condor provides checkpointing facilities [31,30] which, when coupled with the migration facility, allows a job to be transferred to a different resource in case the one on which the job was already executing is no longer free.

2.2. Manually configured flock of Condors

To allow multiple Condor pools to share resources by sending jobs to each other, Condor employs a flocking mechanism [11]. A Condor pool can be statically configured to allow job requests from a known remote pool to be forwarded to it. Flocking works in the following manner. If a pool *A* wants to allow jobs from users in another pool *B* to be run on its resources, the central manager of *A* is configured to allow this sharing. Moreover, the Condor daemons (`condor_schedd`) running on the submission machines in *B* are also explicitly configured to use resources available in *A*. The machines in *B* will only send jobs to *A* if the local resources are unavailable or in use. The job scheduling negotiations for a remote job occur between the submission machine of *B* and the central manager of *A*. Finally, the negotiated job is executed on the remote resource. It should be observed that this mechanism is static, and requires both pools *A* and *B* to be pre-configured for resource sharing.

2.3. *Structured p2p overlay networks*

Structured p2p overlay networks such as CAN [40], Chord [47], Pastry [43], and Tapestry [51] effectively implement scalable and fault-tolerant *distributed hash tables* (DHTs), where each node in the network has a unique node identifier (`nodeId`) and each data item stored in the network has a unique key. The `nodeId`s and keys live in the same namespace, and each key is mapped to a unique node in the network. Thus DHTs allow data to be inserted without knowing where it will be stored, and requests for data to be routed without requiring any knowledge of where the corresponding data items are stored.

The key aspects of these structured p2p overlays are self-organization, decentralization, redundancy, and routing efficiency. Self-organization promises to eliminate much of the cost, difficulty, and time required to deploy, configure and maintain large-scale distributed systems. The process of securely integrating a node into an existing system, maintaining its integrity invariants as nodes fail and recover, and scaling the number of nodes over many orders of magnitude is fully automated. The heavy reliance on randomization (from hashing) in the `nodeId` and key generation provides good load balancing, diversity, redundancy and robustness without requiring any global coordination or centralized components, which could compromise scalability. In an overlay with N nodes, messages can be routed with $O(\log N)$ overlay hops and each node only maintains $O(\log N)$ neighbors.

The functionalities provided by DHTs allow for selecting pools in the presence of dynamic joining and departure of Condor pools. While any of the structured DHTs can be used, we use Pastry as an example in this paper. In the following, we briefly explain the DHT mapping in Pastry.

Pastry: Pastry [44,5] is a scalable, fault resilient and self-organizing p2p substrate. Each Pastry node has a unique, uniform, randomly assigned `nodeId` in a circular 128-bit identifier space. Given a message and an associated 128-bit key, Pastry reliably routes the message to the live node whose `nodeId` is numerically closest to the key.

In Pastry, each node maintains a routing table that consists of rows of other nodes' `nodeId`s which share different prefixes with the current node's `nodeId`. In addition, each node also maintains a leaf set, which consists of l nodes with `nodeId`s that are numerically closest to the present node's `nodeId`, with $l/2$ larger and $l/2$ smaller `nodeId`s than the current node's `nodeId`. The leaf set ensures reliable message delivery and is used to store replicas of application objects. Pastry routing is prefix-based. At each routing step, a node seeks to forward the message to a node whose `nodeId` shares with the key a prefix that is at least one digit longer than the current node's shared prefix. The leaf set helps to determine the numerically-closest node once the message has reached the vicinity of that node.

Pastry takes network proximity into account in building routing tables. It selects routing table entries to refer to nearby nodes, based on a proximity metric, subject to the prefix-matching constraints imposed on the corresponding entries. As a result of the proximity-awareness, a message is normally forwarded in each routing step to a nearby node that is chosen according to the proximity metric out of all the candidate nodes for that hop. Moreover, the expected distance traveled in each consecutive routing step increases exponentially, because the density of nodes decreases exponentially with the length of the prefix match, and the expected distance of the last routing step tends to dominate the total distance traveled by a message. As a result, the average total distance traveled by a message exceeds the distance between the source and the destination nodes only by a small fraction [5].

3. Design

We describe a p2p-based flocking technique that allows a Condor pool to locate one or more dynamically changing remote pools to utilize their resources, and to provide fault tolerance within a pool against central manager failures.

3.1. *Self-organizing Condor pools*

The original flocking scheme has the drawback that the knowledge about all the remote pools with which resources can be shared is required prior to starting Condor, and this information remains static. To overcome this limitation, and to provide self-organization of Condor pools with minimal initial knowledge, we organize the Condor pools using the Pastry p2p overlay network as described in Section 2.3. Pastry arranges the pools on a logical ring—the p2p overlay's node identifier name space—and allows a Condor pool to join the ring using only the knowledge about a single pool that is already in the ring. Once a pool joins the ring, it can reach any other pool on the ring via Pastry overlay routing. The ability to automatically reach all other pools without any initial knowledge about them is enabled by the p2p self-organization of Condor pools.

Another advantage of using Pastry is the automatic creation and maintenance of the proximity-aware routing ta-

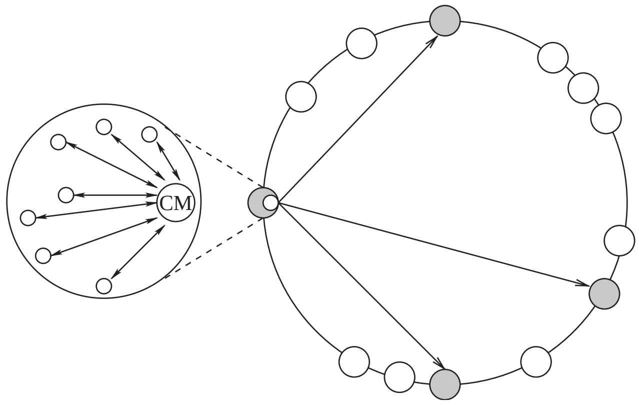

Fig. 1. Interactions among Condor central managers of different pools. The big circle on the right represents the managers arranged in a p2p ring (i.e. the circular node identifier space). Each circle on the ring represents a Condor pool. An enlarged version of a pool is shown on the left. The resources in the pool are only aware of the central manager (CM), and send job requests to it. The CM uses the locality-aware p2p mechanism to determine nearby Condor pools and sends jobs to them if necessary. In the figure, the gray circles indicate the potential remote Condor pools where jobs of the pool on the left can be forwarded.

ble which can be used to sort available remote pools in order of the network proximity. This allows a Condor pool to announce its available resources to various pools in a proximity-aware fashion.

Fig. 1 shows the overall layout of the proposed approach. The enlarged node on the left shows the typical configuration of a Condor pool. The central manager manages the various resources in the pool. These resources can be compute machines that provide computing power, or they can be submit-only machines that act as access points to the shared computing resources. The Condor pools that are interested in sharing resources with other pools form a p2p overlay network, and in doing so each pool is assigned a random node identifier (`nodeId`) in the ring. The `nodeId` is randomly assigned, e.g., as the secure hashing (SHA-1 [12]) of the IP address of the host, and thus nodes that are adjacent in the node identifier space may be far apart in the physical network proximity space and vice versa. For instance, the gray nodes in Fig. 1 are physically close, but are not adjacent in the identifier space. It should be noted that only the central manager needs to be part of this logical ring. Other resources in a pool are not aware of the p2p organization of the pool managers, and continue to interact with the central manager.

The self-organization of the central managers may be affected by high degree of disconnections and joins to the p2p overlay. As only the central managers are part of the overlay, node failures/joins within a pool does not effect the overlay. Such disconnections are managed by Condor and does not pose a problem to the overlay management. Since the central managers are usually chosen to be the machines that are expected to be most stable, we also expect that they will behave in a stable manner in the p2p overlay. That is, the expected number of disconnections or failures will be low for each central manager. Hence, the overlay management is not expected to experience a high degree of nodes leaving or joining the system. Moreover, techniques such as described in [41] can be adopted to ensure acceptable behavior of the p2p overlay in case the degree of failures increases.

3.2. *Proximity-aware remote pool discovery*

Once the pools are self-organized into a p2p overlay, various methods can be adopted to determine which remote pools are most suitable to send jobs to. Without loss of generality, we will refer to the pool that is making this determination as the local pool in the following discussion.

One method is that the local pool broadcasts a query for available resources to all remote pools in the p2p overlay, and chooses to flock to a pool that replies with a willingness to share its resources and is nearby. However, broadcast generates unnecessary traffic if most of the time the available resources can be found from a subset of the pools in the overlay.

A more efficient method is to leverage the p2p overlay for the selection of remote pools. The advantage of using a p2p overlay is that it can help to efficiently locate remote Condor pools. Moreover, utilizing the locality-aware p2p routing guarantees that jobs will not be shipped across long distances in the network proximity space if willing Condor pools are available nearby. To achieve these goals, the locality-aware routing table of Pastry as discussed in Section 2.3 is exploited. We discuss a p2p-based method for the selection of remote pools in the following.

3.2.1. *Basic design*

Each pool that has resources available sends a message announcing the available resources to all the pools specified in its routing table, starting from the first row and going downwards. Thus a pool always contact nearby pools first. On receiving such a message, a pool becomes aware of the nearby free resources, which it can then select for flocking. Such selection of nearby pools translates to saved bandwidth in terms of data transfer that may happen between a job submission machine and the job execution machine, and thus a higher overall job throughput. For instance, Fig. 1 shows the local pool utilizing resources from various gray nodes, which are chosen as the Condor pools that are close to the local pool.

The dynamic resource pool discovery is achieved via a software layer. The software runs on each central manager M and uses the resource announcements from other managers M_R to decide which resource pools to flock to. An announcement from M_R contains information about the available resources in its pool, and its desire to share the resources with M. Note that M_R and M do not need to be statically aware of each other to decide whether sharing with each other should be allowed. They can individually maintain history about pools with whom previous sharing was not beneficial, or can dynamically learn of "black listed" pools from other trusted pools. Finally, an expiration time is also contained in the announcement to inform *M* of the duration the information contained in the announcement is valid for. From this information, *M* can create a list of resource pools that are available to it, ordered with respect to the network proximity. This list is referred to as `willing_list`. It is an array of sublists, with the *i*th sublist containing M_Rs whose *i*th row of the routing table contains *M*. Hence, due to the proximity-awareness of Pastry's routing table, the resources in the first sublist of the `willing_list` are exponentially nearer compared to the resources in the second sublist, and so on. If several resource pools in a sublist share the same proximity metric, the order of these pools is randomized before configuring Condor to use them for flocking. Doing so ensures that if many nearby pools discover the same set of free resources simultaneously, any particular free resource is not overloaded. Finally, the central manager pushes the `willing_list` to the submission machines in the local pool. The submission machines can use this information to enable flocking with resources in remote pools.

3.2.2. Optimization

One potential drawback of the above approach is that the Pastry routing table of a given central manager may only contain information about a subset of all available and nearby pools, i.e., those whose `nodeId`s match the `nodeId` of the central manager in the respective prefixes. This can limit the scope of p2p-based flocking: when all the pools known to the routing table are unavailable due to either lack of free resources or absence of access permissions, a Condor pool *M* will not be able to flock to other pools whose Pastry routing tables do not contain *M*.

To address this problem, the p2p-based flocking can be extended as follows. Instead of propagating the availability information to only the nodes in the routing table, a time-to-live (TTL) field is introduced in the announcement message, so that the message can be propagated to pools several hops away in the overlay network. The TTL is a system-wide parameter, and can be adjusted dynamically to support various load conditions of the whole system. On receiving a message, a pool *M* decrements the TTL, and if the TTL is greater than zero, pool *M* forwards the message to the pools specified in its corresponding routing table row. In this way, the TTL controls how far the resource availability announcement will be propagated. The receiving node creates a list of all the remote pools that are willing to share resources with it. It then probes these pools to determine how far they are, and use this information to generate the `willing_list` that is sorted with respect to the network proximity. Each sublist in this `willing_list` contains nodes that initiated resource announcements using the same row of their routing tables though they may be several overlay hops away. Due to the nature of how the announcements are forwarded, successive sublists contains nodes that are increasingly farther apart.

3.2.3. Discussion

The selection of a remote pool for flocking requires discovering available remote pools with free resources, and knowing the pool's willingness and policy for sharing the free resources. While the p2p-based technique automates locating available remote pools, it retains each individual pool's control of access to its resources. This provides the separation of the resource discovery mechanisms from the sharing policies of individual pools, hence, giving pool owners full control of how their resources are shared. In order words, the p2p-based flocking scheme focuses on resource discovery, and the policy specification is left to the individual central managers.

Same as in the original flocking mechanism, our proposed p2p-based flocking mechanism also decouples the flocking of jobs across different Condor pools from the matchmaking process for scheduling jobs within each pool [37,39]. Matchmaking provides a mechanism for a job to be sent to a suitable resource. Flocking, on the other hand, locates remote pools to which such requests can be forwarded. Matchmaking is locally employed in the remote pool to select a suitable resource. Our proposed scheme periodically receives metrics such as queue lengths, average pool utilization, and the number of resources available from remote pools, and builds a list of available pools with whom flocking can be done. The list is then sorted using the proximity information about the remote pools, from nearest to farthest. This dynamically ordered list is then utilized by Condor on each submitting machine in the pool to select a remote pool to flock to. This is in contrast to the original flocking scheme where the order and number of pools to flock to are static and configured manually.

Alternatively, a direct matchmaking technique can be employed in the following way. Remote pools propagate a set of ClassAds describing the available resources in a pool, in addition to the pool status metrics as described above. The available remote pools are ordered by proximity as before. In this case, however, individual resource ClassAds are also available, enabling matching of local jobs to remote resources directly in the local (job submission) pool. This is in contrast to flocking to remote pools followed by matchmaking, which may require several iterations of flocking and matchmaking before a suitable resource is found. Such direct matchmaking can potentially yield a more efficient scheme, and is a topic of our future research.

Another interesting issue to consider while using p2p technology for resource discovery is how it compares to centralized solutions. Besides the advantages of fault tolerance, scalability and locality awareness, employing a p2p approach provides freedom from the hierarchical organization of distributed resources that is required in most centralized approaches. This allows pools to interact with any remote pool directly to provide more flexibility. Moreover, the overhead of using the p2p approach is very small. The locality-awareness of our p2p substrate ensures that the cost of communication between two nodes over the overlay incurs only 30–40% extra overhead [44] compared to the direct communication between the two nodes over the physical network.

Finally, in Grid environments, the nodes that are chosen as central managers are expected to be quite stable. Such nodes when organized in a p2p overlay will result in a relatively stable system, requiring very little communication to maintain the overlay. In case of the uncommon scenario of instability in the system, techniques such as those described in [41,32] can be employed to manage highly dynamic resources while guaranteeing reliability with acceptable maintenance costs.

3.3. A fault-tolerant Condor pool

The existing design of Condor provides fault tolerance against failure of resources in a Condor pool, but remains susceptible to the failure of the central manager. The dependence of the whole pool on one central manager can be mitigated by utilizing fault tolerance of p2p overlay routing. All the resources in a Condor pool can be arranged on a logical ring, with the `nodeId` of the central manager known to every resource. This ring is local to a pool and does not interact with the logical ring for on-demand flocking. The central manager is the only node that is on both rings. This provides a hierarchical structure of control similar to that of [20]. The central manager periodically informs every resource in the pool of its aliveness. In addition, replicas of the pool configuration and other management information of the central manager are maintained on the K immediate neighbors of the central manager in the node identifier space of the local p2p overlay. In case the central manager fails, the clients detect its absence and send messages with the central manager's `nodeId` as the message key in the p2p overlay. These messages are guaranteed by the p2p routing to arrive at one and only one of the K neighbors of the failed manager, which then takes on the role of the central manager. As a result, the client machines can continue to submit jobs and human intervention is not required, other than correcting the problems with the failed central manager.

3.4. Security

Sharing resources across administrative domains pose security challenges which if not addressed can lead to the compromise of shared resources. In the following, we discuss various mechanisms that provide security in this context, ordered by their complexity and effects on the performance of the system.

The first security mechanism uses authentication for ensuring that users are accountable for their actions. It has the least overhead on system performance as only verification of identity credentials is required. Condor employs authentication of users as well as resources and provides security policy specification [8]. In a single pool, Condor can be set up to run jobs only from the users who have standard authenticated accounts on the resources. In the presented scheme, a policy manager can be used to authenticate remote pools based on some out-of-band credentials. To protect against a malicious remote Condor pool, we employ a policy file to control a pool's interactions. For example, interactions can be limited to with only those remote pools that have been pre-approved by the pool manager. An encryption layer can also be added on top of this to ensure that a malicious remote pool does not pose as a pre-approved pool. The additional authentication features of systems such as Globus [15,17,14,2] can also be leveraged in the proposed design to ensure more secure operations. Although simple, authentication relies on accountability after the damage is done and does not actually prevent malicious behavior [3].

A second security mechanism provides prevention by restricting access of remote jobs to a safe limit. For example, in UNIX-based systems, jobs from anonymous users and users from remote pools can be executed as user `nobody`, hence curtailing the capabilities of malicious users. This mechanism provides better security, but reduces the capabilities of legitimate programs as well. Such mechanisms do remain susceptible to instrumentation attacks as shown in [3,33].

A third security mechanism is to provide a controlled execution environment to overcome problems in previous schemes. This mechanism is very secure. However, it poses a significant performance overhead due to setting up of execution environments. In case of flocking, the jobs from remote pools can be sandboxed using either the Java Virtual Machine [23] or system call tracing as proposed in [3,22], giving a resource fine-grained control over the actions of the jobs.

Finally, numerous previously proposed intrusion detection schemes (for example, [28,27]) can be employed to detect previously unknown malicious behavior. This can provide tight security but at a significant running cost of security monitors and analyzers.

In summary, pools have the freedom to choose mechanisms that suit their security needs and act on these mechanisms accordingly.

4. Implementation

We implemented the proposed scheme by adding a software layer on top of Condor. The software is implemented using the FreePastry [35] implementation of Pastry API [44], and utilizes the flexible configuration control of Condor to dynamically modify the flocking behavior of Condor.

The main software is divided into two independent components: *poolD* which runs only on the central managers to maintain the self-organized flock and to discover remote

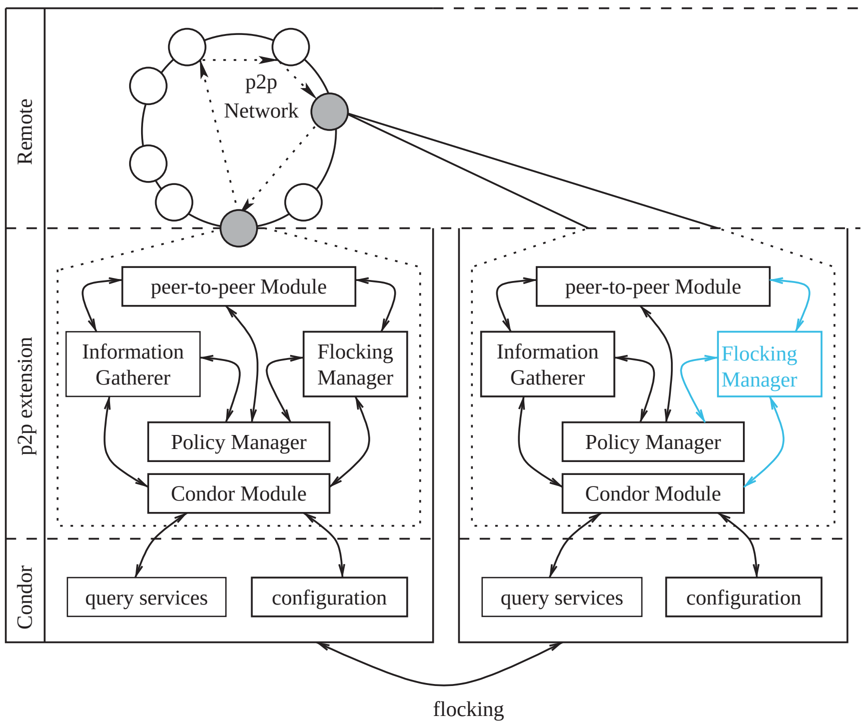

Fig. 2. Architecture of *poolD* and interactions of various modules. *poolD* runs on the central manager of each pool that intends to participate in the self-organizing flocking for sharing remote resources. The resource announcements from the local *poolD* (shown on the left) are received at the `Information Gatherer` in the remote *poolD* (shown on the right). Note that the local *poolD* does not directly interact with the remote `Flocking Manager` (shown in gray).

Condor pools, and *faultD* which runs on all the resources in a Condor pool to provide resilience to central manager failures.

4.1. *poolD*

Fig. 2 shows the various modules of *poolD*, and how it interacts with Condor to control the flocking behavior. It runs on the central manager of each pool where sharing with remote Condor resources is desired.

The `peer-to-peer Module` provides the self-organization features of the scheme. It also takes care of p2p routing and provides a communication facility for sending and receiving messages between the central managers of the pools in the overlay. Other modules in *poolD* can then use this facility to exchange information with their remote counterparts. In addition, the `peer-to-peer Module` performs the task of forwarding resource announcements if TTL is greater than one.

The `Condor Module` provides an interface to the Condor software running on the node. It uses the Condor querying and configuration facilities to obtain runtime information about the local pool and to dynamically configure its behavior.

The periodic update of the `willing_list` is performed as follows. For this discussion, Condor central managers that have joined the p2p ring are referred to as nodes, the node on which the `willing_list` is being constructed is called the local node *L*, and the nodes that announce resource information are called the remote nodes.

The `Information Gatherer` is responsible for sending the resource availability announcements to inform nearby nodes, and also for updating the local `willing_list` on receiving such announcements. Consider a remote node *R*. Whenever resources become available on *R*, an announcement is created as follows. The `Information Gatherer` on *R* periodically contacts its `Condor Module` to obtain the status of the pool. Next, the `Information Gatherer` consults its `Policy Manager`—a module that implements pool sharing preferences—to determine what resources can be shared with which remote pools. The `Policy Manager` utilizes a policy file for this purpose. The policy file itself is a list of machines from which jobs are either permitted or denied. This can be captured by either using explicit machine or domain names, or use of wild cards. After policy verification, the next step is the selection of a TTL and a suitable expiration interval for the availability announcement. Finally, the `Information Gatherer` sends the pool status information along with other bookkeeping information such as announcement expiration time to all the nodes in the Pastry routing table.

This information is received at *L*, and passed to its `Information Gatherer` which first consults the local `Policy Manager`. The `Policy Manager` ensures that individual pools have control over the remote pools of resources on which their jobs are run and from which remote pools the remote jobs are allowed to run locally. If the `Policy Manager` on *L* permits information exchange with *R*, the `Information Gatherer` on *L* updates *L*'s `willing_list`. Otherwise, there is no update to *L*'s `willing_list`. In either case, the announcement is forwarded in accordance with the TTL. The `willing_list` is sorted with respect to proximity. This is done by pinging the nodes on the list and determining their distances from *L*.

Independent of the process for updating the `willing_list`, the `Flocking Manager` on *L* periodically queries the local `Condor Module` to determine if the load on the pool is exceeding the available resources, hence requiring flocking in order to increase throughput. If flocking is required, the `Flocking Manager` examines the `willing_list` and uses the network proximity information to create a sorted list of Condor pools with resources that can be leveraged. If the proximity is the same for multiple pools, the number of free resources available in these pools is taken into consideration, with the pool with more resources given precedence over others. The `Flocking Manager` then uses the `Condor Module` to inform the local Condor central manager of the machines with whom to flock. Similarly, if flocking is enabled, but the `Flocking Manager` determines that local pool is underutilized, it disables flocking. In this way, the various modules interact to maintain a self-organized flock of Condor pools.

4.2. *faultD*

Fig. 3 depicts the architecture of *faultD*. It runs on each resource that is part of a Condor pool, and ensures that the central manager or one of its replicas is always reachable. *faultD* creates another p2p ring comprising of the central manager and all the resources in the pool. It has dual roles: on the submit or compute machines it acts as a passive *Listener*, whereas on the central manager it acts as an active *Manager*. Fig. 4 shows the protocol followed for switching between the two roles. The same software starts on all the resources and the central manager as a *Listener*. Whether a *faultD* is running on the original central manager is determined from a command line configuration parameter. For the original central manager it is specified as `true`, and for every other resource it is either specified as `false` or not specified. The respective roles on various resources are then adopted according to the protocol.

The `Communication Module` is responsible for all the communication between the nodes. It utilizes the Pastry API to route messages between the nodes.

As a *Manager*, *faultD* uses the `Replication Module` to maintain replicas of necessary files on its immediate neighbors in the node identifier space. The `Replication`

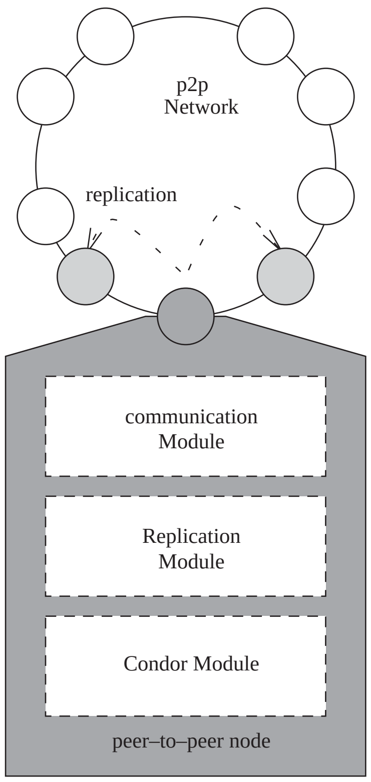

Fig. 3. Architecture of *faultD*. It runs on all the resources in a Condor pool, and provides resilience to central manager failures.

`Module` periodically pushes the up-to-date information to the neighboring nodes to ensure that a backup node with the necessary information is available in case of failure of the central manager. Another task of *faultD* as a *Manager* is to periodically broadcast an `alive` message to all the resources in the pool. The message also contains bookkeeping information such as the `nodeId` of the *Manager*, a monotonically increasing sequence number to detect duplication, and the expected time till the next `alive` message. This information is used by the resources to detect a failure at the central manager. If the original central manager is brought on-line, i.e., it rejoins the local p2p ring, in the presence of an active replacement central manager, the original manager will receive the broadcast `alive` from the replacement and learn the replacement's `nodeId`. The original manager then uses this `nodeId` to send a `preempt_replacement` message to the replacement manager. On receiving this message, the replacement manager transfers the up-to-date pool configuration to the original manager, forfeits its role as the central manager, and becomes a *Listener*.

As a *Listener*, *faultD* passively listens to the `alive` messages from the central manager. Each message is processed to determine whether it is coming from the known central manager or not. In case it does, no further action is required. However, if the message is from a new node, the `Condor Module` is used to update the local Condor to use the new node as the central manager. If the messages

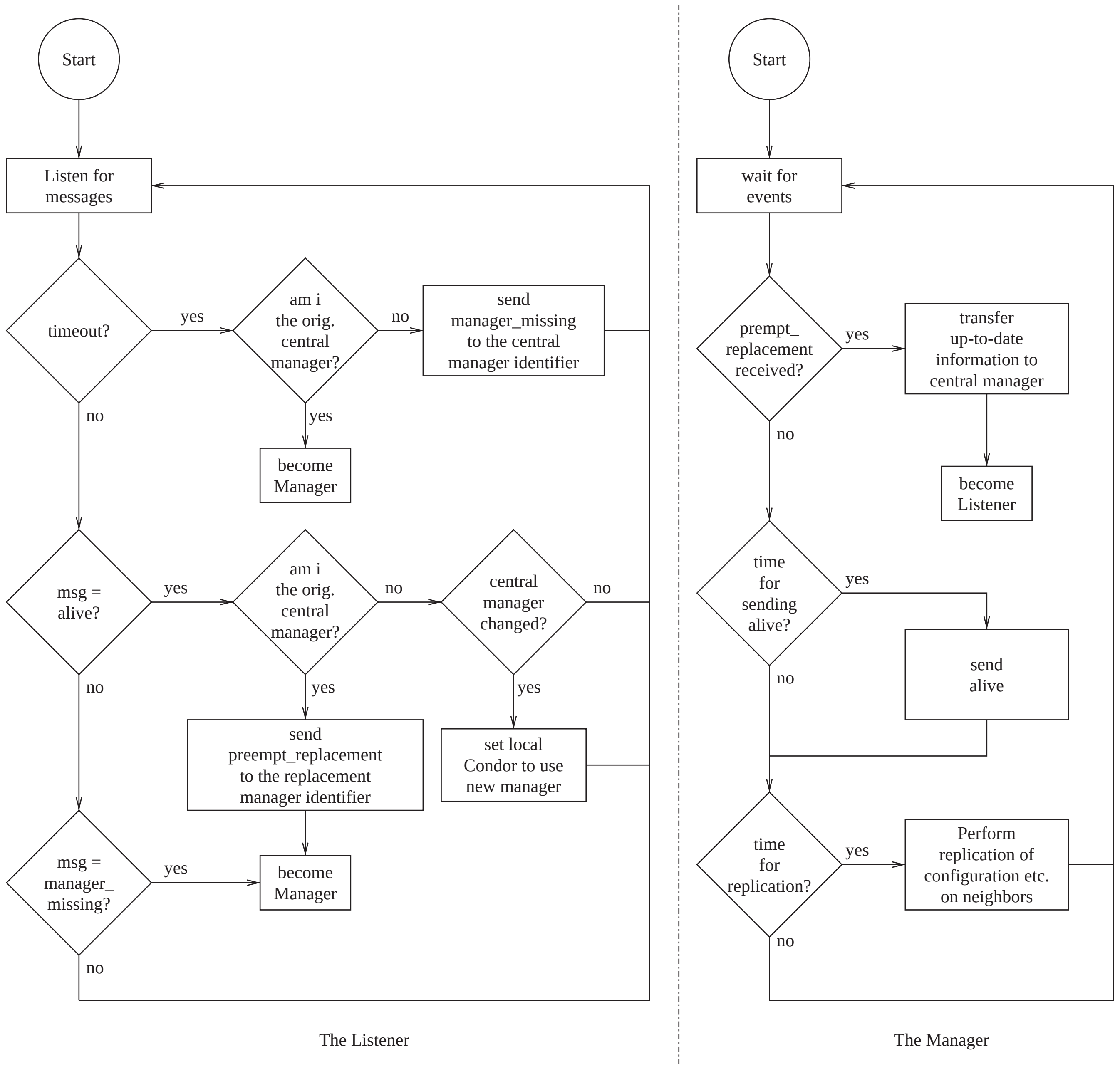
Start
Listen for messages
timeout?
yes
am i the orig. central manager?
no
send manager_missing to the central manager identifier
no
yes
become Manager
msg = alive?
yes
am i the orig. central manager?
no
central manager changed?
no
no
yes
yes
send preempt_replacement to the replacement manager identifier
set local Condor to use new manager
msg = manager_ missing?
yes
become Manager
no
The Listener
Start
wait for events
prempt_ replacement received?
yes
transfer up-to-date information to central manager
become Listener
no
time for sending alive?
yes
send alive
no
time for replication?
yes
Perform replication of configuration etc. on neighbors
no
The Manager

Fig. 4. The protocol followed by *faultD* module to switch between the roles of *Listener* and *Manager*.

stop, the node sends a `manager_missing` message to the previously known `nodeId` of the central manager in the p2p overlay. The p2p routing guarantees that this message will be delivered to either the central manager (if it is alive) or one of its immediate neighbors whose `nodeId` is closest to the central manager's `nodeId` in the node identifier space (if the central manager is no longer available). The detecting node then goes back to the listening state.

If a *Manager* receives a `manager_missing` message, suggesting its `alive` message to a specific node was lost, it simply ignores this message and continues to send `alive` messages. The node that could not receive the message previously will receive this message and will continue to operate normally as described above.

If a *Listener* receives a `manager_missing` message, it implies that the central manager has failed, and that the receiving node is the nearest node to the failed manager in the node identifier space of the p2p overlay. Consequently, the receiving node is the replacement manager. In this event, *faultD* takes on the role of the *Manager*. There are two aspects that allow the node to assume the role of a central manager. One, it already has the replicated pool configurations which allows it to perform the duties of the central manager. Second, job queues are maintained on the submission nodes (not on central manager) and once matchmaking is done and the job starts to execute, the central manager is out of the submission-execution loop till either the job completes or is checkpointed and needs to be migrated to a new resource. Therefore, failure of central manager does not affect the running jobs. The failure does affect the job requests for which the matchmaking is in progress at the time of the failure. This is not a problem because of the following sequence of events. Once the replacement manager becomes active it informs all the resources in the pool about itself. The resources now send their ClassAds to the replacement manager. Similarly, the submission machines resend the job requests to the replacement manager. Finally, with the information about resources and outstanding job requests, the replacement manager can restart the matchmaking. Hence, transition to a replacement manager does not cause loss of jobs.

5. Evaluation

We have extended Condor Version 6.4.7 with the self-organization capability described in this paper. In the following, we report both the experimental results of our extended Condor system on PlanetLab [36] as well as the results from simulating a large number of Condor pools over the Internet. Note that compared to the results presented in our earlier work [4] where we utilized pools in a local area setting, this work presents more thorough experiments conducted over the Internet using PlanetLab.

5.1. Measured performance results

The purpose of the measurements is to determine:

- The effect of flocking on job throughput compared to without flocking.
- The job throughput achieved by flocking among several pools compared to an integrated pool containing the same number of compute machines as in the distributed pools. The performance of the integrated pool gives an upper bound on the job throughput of a fixed set of machines.

5.1.1. Methodology

In order to perform the measurements, we utilized the wide-area testbed provided by PlanetLab [36]. We chose four sites across the United States and Europe to host four Condor pools. Table 1 shows the chosen sites and their geographic locations. At each site there are three machines each available for computations. The configurations are shown in Fig. 5.

In order to drive our measurements, we required a trace that provides the issue time of individual jobs. In Condor, the jobs are queued and maintained on the submission machines and the central manager is only informed of the queued jobs when a resource is required to run these jobs. A job may stay on a local queue for some time before the central manager becomes aware of its existence. No central job issue time statistics are maintained, and it would require modifying Condor on all submission machines to collect such a trace. Because of these challenges in collecting a real job trace, our attempt to do so was unsuccessful.

Table 1
The PlanetLab sites chosen to run the Condor pools, and their locations

Site letter	Location	IP address (CM)	Num. of machines
A	Interxion, Germany	80.253.103.41	3
B	University of California, Berkeley	169.229.51.250	3
C	Columbia University, NY	128.42.6.145	3
D	Rice University, TX	128.59.67.202	3

Alternatively, to measure the effects of various configurations on the scheduling of jobs and the resulting throughput, we created a synthetic job that would consume resources for any specified amount of time. The simple idea of using the standard busy-wait loop that can be used to consume time on a standard pool cannot be utilized in the context of PlanetLab, as that would unnecessarily load the remote machines and violate the PlanetLab acceptable-use policy. Therefore, our synthetic job simply uses the `sleep()` system call and consumes time while avoiding actually over-loading the execution resource.

We then created a sequence of 100 submissions of the synthetic job, each with a random duration between 1 and 17 min, issued with a random interval between 1 and 17 min, with an average of 9 min. The choice of these ranges is arbitrary with the objective of creating jobs that are not unnecessarily long for the experiments, yet exhibit interesting behavior. We created 12 such job sequences, enough to keep 12 machines busy all the time. For the case of four separate Condor pools, the 12 job sequences are merged into four different job traces, one for each pool. A job trace with n job sequences merged together implies that it on average has n job requests issued simultaneously. The number of sequences in the four job traces were 2, 2, 3, 5 for pools A, B, C, and D, respectively. For the case of a single integrated pool, we merged all 12 sequences into a single trace.

In order to use the generated job traces, we implemented a job driver which takes as input the traces, and submits the specified length synthetic jobs to the respective Condor pools at specified times. The pools were set up so that jobs would start running on any of the compute machines or the central manager if available, hence giving a total of three compute resources per pool. The machines were dedicated to these jobs, therefore the effects of checkpointing because of an owner returning to the desktop were avoided. The TTL parameter and the expiration interval in availabil-

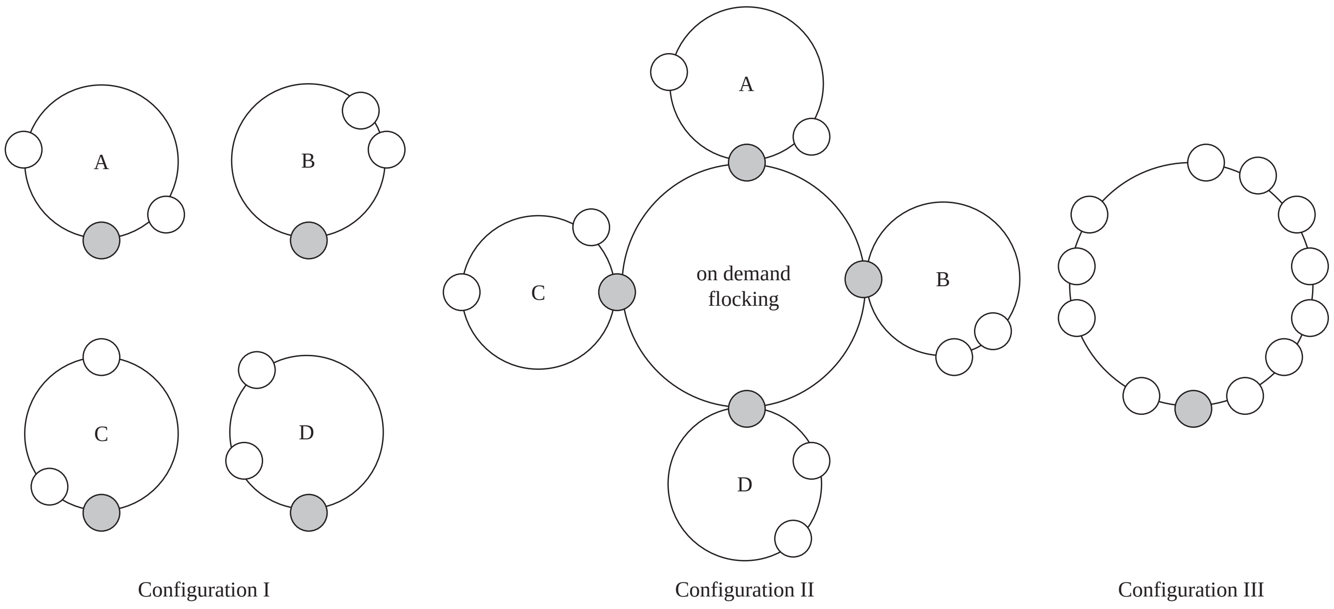

Fig. 5. The various configurations of Condor pools used for making performance measurements. (I) Individual pools driven by local job traces, (II) Pools in Configuration I connected via on-demand flocking, and (III) All machines arranged in a single pool across administrative domains. Each ring represents a p2p overlay network of nodes. The gray nodes are the central managers and the white nodes are the compute machines.

ity announcement messages were set to one and 1 min, respectively. The interval at which the `Flocking Manager` queries the local `Condor Module` in *poolD* was also set to 1 min.

5.1.2. Results

Tables 2 and 3 show the wait times of jobs in the queue for the configurations in Fig. 5. In Configuration I, we fed the traces to individual pools without flocking. The number of sequences in the job traces varied from 2 to 5, whereas the number of computing machines in each pool was fixed at 3. It was observed that jobs have to wait in queue for as long as 284.91 min on average in pool D, during which time machines in pool A were idle. Also note that at least one job in pool D was in queue for a huge period of 557.55 min.

Next, we measured how on-demand flocking can utilize the multi-pool resources with Configuration II. Here we used the same individual pools of Configuration I, but ran the p2p flocking software on each central manager to facilitate self-organized flocking. The pools were driven with respective job traces as in Configuration I. It was observed that compared to 284.91 min in Configuration I, the average wait time for pool D was reduced to 28.37 min. In addition, the maximum wait time with flocking is reduced by a factor of 9.55 compared to without flocking. Pool C has a small improvement in wait time over that in Configuration I. The reason for this is that pool C has three machines and is driven by a job trace with an average issue of three jobs at a time. Therefore, pool C does not provide its resources to other pools. The improvement in average job wait time of pool C is due to the fact that at peak loads, it was able to utilize the machines in other pools. The effect on pools A and B is an increase in average queue wait times. This is because pools A and B are now sharing resources with heavily loaded pools, such as pool D. At times, Pool A would be idle and jobs from pool D would start running on it. Then if a job is issued at pool A, it had to wait for the jobs from pool D to finish, since in these experiments pools are configured not to suspend and move a job once that job has started executing. Note, however, this is a matter of policy, and the local pool can be set up with difference policies. Compared to Configuration I, the overall mean wait time is reduced by 100.9 min, whereas in pools A and B, it is increased by only 18.39 and 29.38 min, respectively.

Next, we determine how the wait times of jobs will change if all the machines were available in a single pool. For this purpose, we merged the machines into a single pool with 12 compute machines (Configuration III), and loaded the pool with a trace with all 12 sequences. In this configuration, the average amount of time the jobs had to wait in the Condor wait queue before being scheduled was only 27.47 min. This shows the efficiency of a combined large pool in scheduling jobs. However, merging the machines across administrative domains is not a desirable approach to improving the throughput, as such merging requires administrative privileges across organizational boundaries [11].

To determine how flocking affects the wait times of jobs when compared to the single integrated pool, we loaded Configuration II at one of the pools (D) with the same job trace with 12 sequences as used to load the single pool of Configuration III. The results show that the wait times in

Table 2
Wait times for jobs in queue for Configurations I and II

Pool	No. of seq.	Without flocking (Conf. I)				With flocking (Conf. II)			
	in job trace	mean	min	max	stdev	mean	min	max	stdev
A	2	1.76	0.03	14.32	2.65	20.15	0.03	72.10	23.84
B	2	3.30	0.08	19.85	3.84	32.68	0.13	63.70	17.74
C	3	46.58	0.03	97.17	23.00	38.68	0.10	64.48	16.58
D	5	284.91	0.25	557.55	178.94	28.37	0.10	58.38	25.38
Overall	12	131.20	0.03	557.55	175.01	30.30	0.03	72.10	23.19

All numbers are in minutes. One job sequence contains 100 jobs of random length of 1–17 min, issued at random intervals between 1 and 17 min.

Table 3
Wait times for jobs in queue for Configurations III and II with loading at D

	No. of sequences in job trace	mean	min	max	stdev
Single Pool (Conf. III)	12	27.47	0.03	55.62	18.40
Conf. II (all load at D)	12	27.40	0.03	54.80	18.51

All numbers are in minutes. One job sequence contains 100 jobs of random length of 1–17 min, issued at random intervals between 1 and 17 min.

the two scenarios are almost the same. The few seconds difference is due to the fact that in case of flocking, jobs are first sent to local resources, and then to nearby pools, hence leveraging locality. On the other hand, for a single pool, jobs are distributed to any available resource regardless of its proximity to the issuing machine. This introduces overhead due to shipping a job to far away nodes.

We further compare concurrent job loading at multiple pools versus job loading at a single pool on job wait times with flocking (Configuration II vs. Configuration II, all load at D). Tables 2 and 3 show that the difference is insignificant. The 2.90 min difference in the mean times can be explained by the observation that when individual pools are loaded, preference is given to local jobs for scheduling, which may not be the best method for global scheduling of jobs over all the pools. Moreover, in individually loaded pools, four jobs may be processed simultaneously (one each by each Condor manager) compared to just a single job processed by the single Condor manager. This adds to the process of job negotiation between multiple negotiators, i.e., jobs are first sent to local managers, which then negotiate the jobs with remote managers (which may be processing their local jobs and hence do not reply immediately), and finally the job are sent to the chosen remote resource. This potentially lengthens the overall submission to execution time.

In summary, these results show that without requiring resource merging, the self-organizing flocking mechanism presented in this paper cannot only achieve a significant improvement in job throughput over without flocking, but also achieve a comparable performance to that of a single integrated pool, which is not practical because of issues involved with crossing multiple administrative domains.

5.2. Simulation results

This section presents results of simulating a large number of distributed Condor pools that implement the proposed p2p-based flocking scheme.

5.2.1. Methodology

For the purpose of these simulations, a router network was generated by GT-ITM using the transit-stub model [50]. The IP network consists of 1050 routers, 50 of which are used in transit domains and the rest 1000 in stub domains. The routing policy weights generated by the GT-ITM generator are used to calculate the shortest path between any two nodes. The length of this path allows us to determine the physical "closeness" of the two nodes.

We assume that there is one Condor pool attached to each stub domain router, giving us a total of 1000 pools. The sizes of simulated Condor pools are uniformly distributed between 25 and 225 machines. Following the proposed flocking scheme, the 1000 central managers from these simulated pools form a p2p overlay network using Pastry.

As in the case of prototype measurements, a synthetic job trace is created to drive the simulations. As before, a single job sequence consists of 100 jobs. The inter-arrival time between any two consecutive job requests follows a random uniform distribution between 1 and 17, giving an average of 9 time units. The lengths of jobs also follow a random uniform distribution between 1 and 17 time units. At each Condor pool, one job trace is created to drive the simulations by merging a random number of such single job sequences. The number of single sequences per job trace follows a uniform distribution between 25 and 225.

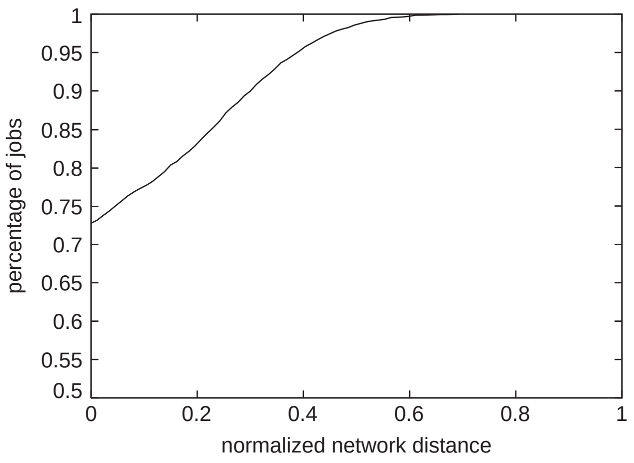

Fig. 6. Cumulative distribution of locality for scheduled jobs when flocking is enabled. The *x*-axis stands for the ratio between the network distance from the job submission pool to the actual execution pool and the diameter of the underlying IP network. Locality zero means that the jobs are scheduled inside local Condor pools.

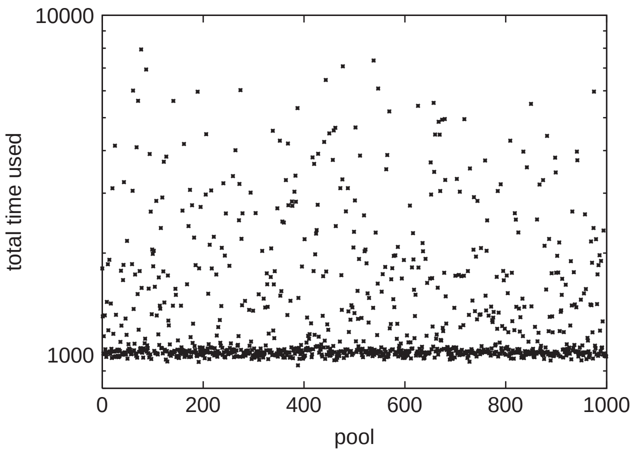

Fig. 7. Total completion time at each Condor pool without flocking.

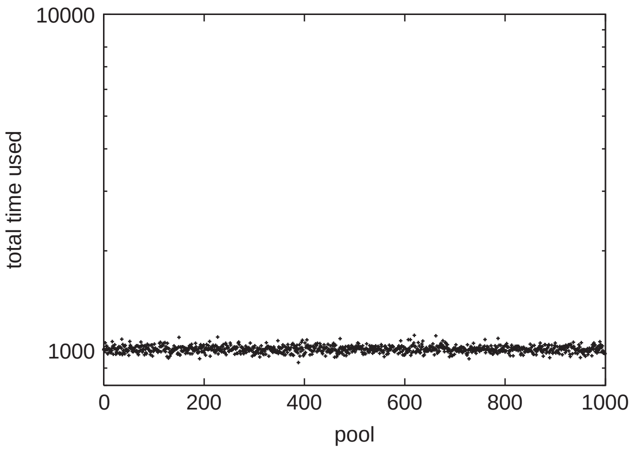

Fig. 8. Total completion time at each Condor pool when flocking is enabled.

A Condor manager attempts to schedule a job request to the machines in the local pool and invokes the flocking mechanism only if all the local machines are busy. Job requests are queued if they cannot be scheduled immediately and each queue is maintained as a FIFO. The simulations are considered complete when all the job requests have been issued and the queue at every central manager is emptied. As in the prototype measurements, the TTL parameter and the expiration interval in the availability announcement messages were set to one and one time unit, respectively, and the interval at which the `Flocking Manager` queries the local `Condor Module` in *poolD* was also set to one time unit.

5.2.2. *Results*

Three sets of simulation results are presented. First, we measured the effectiveness of locality-aware p2p routing in discovering nearby resources to execute jobs. Fig. 6 shows the cumulative distribution of the locality for the jobs scheduled by self-organized flocking. The distance was measured as the network routing delay between the pool where the job is submitted and the pool where the job is scheduled to execute, and represents the locality of a scheduling. This distance is further normalized by the diameter of the underlying IP network (from the GT-ITM generator). The simulations show that more than 70% jobs are scheduled inside local Condor pools and the rest of the jobs are flocked to pools that are close in terms of network proximity. For instance, over 80% jobs are scheduled to pools that are within 20% of the network diameter, over 95% are scheduled to pools that are within 35% of the network diameter, and no jobs travel more than a distance of 70% of the diameter of the underlying network.

In the second set of results, we measured the effects of flocking on the total completion time for all the jobs. We measured the total time units used to complete executing all the jobs. Fig. 7 shows the total time that it takes to complete all the jobs without flocking, observed at each Condor pool. Similarly, Fig. 8 shows the total completion time when self-organizing flocking is enabled. As the figures show, in the absence of flocking, the time required to complete executing jobs at individual Condor pools may vary significantly, and some Condor pools need much more time than others. On the other hand, flocking can evenly distribute workloads among all the available resources, and hence executing jobs at each Condor pool takes about the same amount of time and all the job queues are emptied almost f simultaneously.

In the third set of results, we measured the effects of flocking on the average wait time of jobs in the job queue. The wait time in the job queue is the duration between the time unit that a job is issued and the time unit that the job is dispatched from the queue. Fig. 9 shows the average wait time in queue without flocking, and Fig. 10 shows the same when self-organized flocking is utilized. The simulation shows that

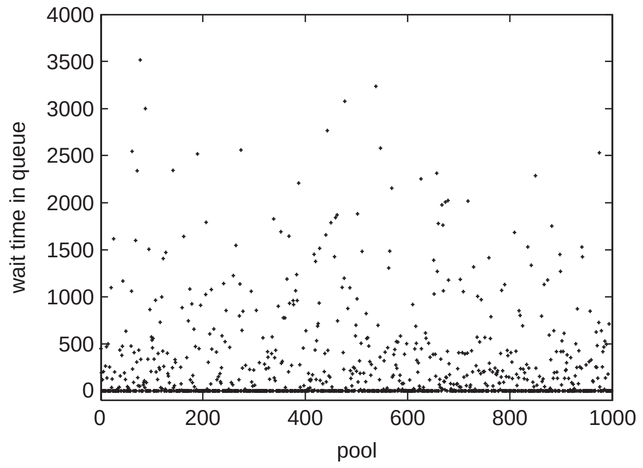

Fig. 9. Average wait time in the job queue at each Condor pool without flocking.

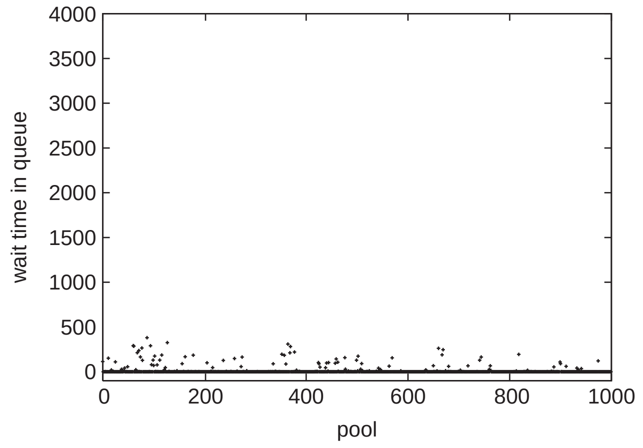

Fig. 10. Average wait time in the job queue at each Condor pool when flocking is enabled.

flocking can significantly reduce the average wait time of a job request in the job queue. Without flocking, jobs in heavily loaded pools have to wait in the queue for a long period, while at the same time machines are idle in lightly loaded pools. This wait time is as high as 3500 time units. When flocking is employed, the maximum wait time remains under 500 time units.

In summary, we have shown that the self-organization of Condor pools provides a scalable and flexible method of utilizing the flocking mechanisms supported by Condor. Our approach increases the opportunity for flocking in a decentralized manner, which results in an increased number of available resources for running the submitted jobs.

6. Related work

In this section, we review related work on resource discovery and management within and across multiple administrative domains, and recent p2p approaches to resource discovery and management.

6.1. Resource discovery and management

Resource discovery is concerned with location, allocation, and authentication of resources. Resource management implies preparation of a resource for use, monitoring for performance, and eventual tear down of execution environment when the job completes or migrates to some other resource. Note that the scheduling, decomposition, assignment, selecting execution order of tasks, and management of low-level resources such as memory, disk, and networks, are not part of resource discovery and management components and are handled separately as needed.

LoadLeveler [26] is a resource management system that handles homogeneous resources in a parallel computer. It is centralized, but does support co-allocation, which is the capability to allocate multiple resources simultaneously while satisfying multiple resource requirement constraints.

In contrast, NQE [9], LSF [53], I-SOFT [13], and portable batch system (PBS) [1], propose network batch queuing systems that focus on a set of network connected computers rather than a parallel computer. In general, these systems rate poorly in handling heterogeneous substrates, and provide only limited on-line control and resource co-allocation. They do not support dynamic extensibility to resource sharing and utilization policies. Typically, these systems process user-submitted jobs by finding resources that have been identified either explicitly through a job control language or implicitly by submitting the job to a particular queue that is associated with a fixed set of resources. Load-balancing is not done automatically across queues, and manual specification hinder the dynamic resource discovery.

Resource discovery in Condor [29] is based on the *Classified Advertisement* (ClassAd) mechanisms [37]. As the system is centralized in nature, each resource is aware of the central manager responsible for resource management in the pool. The resource generates a ClassAd, specifying its nature, preferences, as well as constraints under which a remote job can be executed on it. The ClassAd is received at the central manager, which then acts as a matchmaker for queued jobs and the available resources. Once a suitable match is made, the matched entities are informed of the match, and the resource matching completes. The actual allocation of the resource is then done by Condor, and is followed by job execution. This scheme allows Condor to leverage a variety of matchmaking policies as discussed in [38].

Compared to local-area network schemes discussed so far, resource discovery and management are more challenging in wide-area networks, where resources may span multiple administrative domains. Legion [7] provides a resource discovery infrastructure that supports heterogeneous resources distributed over such a wide-area setup. It utilizes a static task graph for resource allocation, and can provide advance reservations of resources. An important contribution of Legion is that it decouples local resource management from global management. Hence it supports resource discovery, dynamic resource status monitoring, resource allocation, and job control. However, the features are available to only those applications that can access the Legion object-oriented programming model. Legion can also provide management for PVM [48] message passing library based programs. Another example of a similar system is Gallop [49] which is also capable of managing resources in a wide-area network.

A more formal wide-area resource sharing model is adopted in the computational grid [16], which is a collection of geographically distributed hardware and software resources (typically spanning multiple administrative domains) that are made available to groups of remote users.

Globus [15] provides a grid infrastructure to support resource sharing across multi-administrative domains. It has extensive security support for resource and user authentication. Globus utilizes a hierarchical system to discover resources, and to assemble the resources for use as collections of computational nodes on as-needed basis. A prime objective in grid systems such as Globus is to make the physical location of resources transparent to the user. It employs an extensible resource specification language (RSL) and distributed resource allocation managers (GRAMs) to match jobs to resources.

6.2. p2p approaches to resource management

Recently, several efforts have looked at decentralizing grid services using unstructured overlay networks. In [25], Iamnitchi et al. argued for a decentralized solution to resource discovery in grid environments that organizes resource management nodes into an unstructured p2p overlay network and resource requests are forwarded among the nodes if they cannot be satisfied by a given node. They also presented and evaluated different non-flooding-based request forwarding strategies in various resource sharing environments. In [42], Ripeanu and Foster described a decentralized, adaptive mechanism for replica location in wide-area distributed systems. Unlike traditional, hierarchical and more recent p2p distributed search and indexing schemes, replica location nodes in this mechanism do not route queries. Instead, they organize into an unstructured p2p overlay network and replicate location information. The authors argued that this approach generates comparable traffic as structured-overlay-based approaches for data intensive applications with a few thousand location nodes and with query rates being an order of magnitude higher than replica addition/deletion rates.

Several recent works utilize structured p2p systems for resource discovery. Similar to our approach, XenoSearch [46] also utilizes p2p facilities of Pastry [44] for location of resources. Here, the nodes form a self-organized system, and information about resources is partitioned among nodes. Queries for specific resources are directed to the node responsible for the partition. Replication of resource data is also employed, and the most appropriate replica according to the policies of the searching node is utilized. The use of multi-dimensional search to simultaneously match multiple required resource attributes with those in the job specification provides an efficient means for locating resources.

A *ticket*-and-*lease* based advance reservation of resources layered on p2p mechanisms is proposed in SHARP [19]. It manages a set of constrained resources in an efficient way similar to the airline reservation system. Users can reserve resources in advance by obtaining *tickets*, which are probabilistic guarantees that resource will be available when desired. However, when a resource is available, the user is issued a *lease*, which provides a hard guarantee of resource availability. In this way, users can overbook resources, but will only have access to those for which they hold *leases*. SHARP is more focused on managing available resources than on discovering new ones. The goal of this work is to develop a simple, robust, and decentralized technique for sharing (discovery and allocation) resources using the p2p technology. It can be extended into the grid platforms for scalable, distributed resource discovery.

7. Conclusions

We have presented a locality-aware peer-to-peer based approach to remote Condor pool discovery, which yields a self-organizing flock of Condor pools. The previous static flocking mechanisms available in Condor provide a means for sharing resources across pools, but are not suitable in a dynamic and large-scale scenario where different pools have different sharing and utilization preferences. The p2p technology provides a suitable substrate for resource discovery, as it is well suited to a dynamic environment. Moreover, p2p mechanisms are scalable, robust, and fault-tolerant. The locality-aware routing used in the proposed scheme has an added advantage that resources nearby in the physical network are utilized. This translates to saved bandwidth by avoiding data transfer to far away locations, and thus yields a higher job throughput. The self-organization of Condor pools provides a scalable and flexible method of utilizing Condor flocking via dynamic discovery of remote Condor pools. Measurements of our prototype implementation running on four Condors pools with a total of 12 computing machines on PlanetLab, driven by a synthetic job trace, have shown that for heavily loaded pools, the self-organizing flocking can reduce the maximum job wait time in the queue by a factor of 10 compared to without flocking. Simulations of 1000 Condor pools have shown that locality-aware routing indeed leads to flocking with physical nearby pools most of the time. These results show that p2p technology offers a promising approach to dynamic resource discovery essential to high throughput computing.

Acknowledgments

We thank Miron Livny for his many helpful comments on the project. This work was supported by an NSF CAREER award (ACI-0238379).

References

[1] A. Bayucan, R.L. Henderson, C. Lesiak, B. Mann, T. Proett, D. Tweten, Portable batch system: external reference specification, Technical Report, MRJ Technology Solutions, 2672 Bayshore Parkway, Suite 810, Mountain View, CA 94043, ⟨http://pbs.mrj.com⟩ July 2005.

[2] R. Butler, D. Engert, I. Foster, C. Kesselman, S. Tuecke, J. Volmer, V. Welch, A national-scale authentication infrastructure, IEEE Comput. 33 (12) (2000) 60–66.

[3] A.R. Butt, S. Adabala, N.H. Kapadia, R.J. Figueiredo, J.A.B. Fortes, Grid-computing portals and security issues, J. Parallel Distrib. Comput.: Special issue Scalable Web Services Archit. 63 (10) (2003) 1006–1014.

[4] A.R. Butt, R. Zhang, Y.C. Hu, A self-organizing flock of Condors, in: Proceedings of the ACM/IEEE SC2003, Phoenix, AZ, 2003.

[5] M. Castro, P. Druschel, Y.C. Hu, A. Rowstron, Exploiting network proximity in peer-to-peer overlay networks, Technical Report MSR-TR-2002-82, Microsoft Research, 2002.

[6] M. Castro, P. Druschel, A.-M. Kermarrec, A. Rowstron, Scribe: a large-scale and decentralised application-level multicast infrastructure, IEEE J. Selected Areas Commun. (JSAC) (Special issue Network Support Multicast Commun.) 20 (8) (2002) 100–110.

[7] S. Chapin, D. Katramatos, J. Karpovish, A. Grimshaw, Resource management in legion, in: Proceedings of the Fifth Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP'99), San Juan, Puerto Rico, 1999.

[8] Condor Team, Condor Version 6.4.7 Manual, Technical Report, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI, 2003.

[9] Cray Research, NQE User's Guide, Technical Report 007-3794-001, Cray Research, ⟨http://www.cray.com/craydoc/20/manuals/2148_3.3/2148_3.3-manual.pdf⟩ July 2005.

[10] F. Dabek, M.F. Kaashoek, D. Karger, R. Morris, I. Stoica, Wide-area cooperative storage with CFS, in: Proceedings of the 18th ACM Symposium on Operating System Principles (SOSP'01), Chateau Lake Louise, Banff, Canada, 2001, pp. 202–215.

[11] D.H.J. Epema, M. Livny, R. van Dantzig, X. Evers, J. Pruyne, A worldwide flock of Condors: load sharing among workstation clusters, Future Generation Comput. Systems 12 (1) (1996) 53–65.

[12] FIPS 180-1, Secure Hash Standard, Technical Report Publication 180-1, Federal Information Processing Standard (FIPS), NIST, US Department of Commerce, Washington DC, April 1995.

[13] I. Foster, J. Geisler, B. Nickless, W. Smith, S. Tuecke, Software infrastructure for the I-WAY high-performance distributed computing experiment, in: Proceedings of the Fifth IEEE Symposium on High Performance Distributed Computing (HPDC-5), Syracuse, NY, 1996.

[14] I. Foster, N.T. Karonis, C. Kesselman, S. Tuecke, Managing security in high-performance distributed computations, Cluster Comput.: J. Networks Software Tools Appl. 1 (1) (1998) 95–107.

[15] I. Foster, C. Kesselman, Globus: a metacomputing infrastructure toolkit, Internat. J. Supercomput. Appl. High Perform. Comput. 11 (2) (1997) 115–128.

[16] I. Foster, C. Kesselman (Eds.), The GRID: Blueprint for a New Computing Infrastructure, Morgan Kaufmann Publishers, Los Altos, CA, 1999.

[17] I. Foster, C. Kesselman, G. Tsudik, S. Tuecke, A security architecture for computational grids, in: Proceedings of the Fifth ACM Conference on Computer and Communication Security (CCS 98), San Francisco, CA, 1998, pp. 83–92.

[18] J. Frankel, T. Pepper, The Gnutella protocol specification v0.4 (2000), ⟨http://cs.ecs.baylor.edu/∼donahoo/classes/4321/GNUTellaProtocolV0.4Rev1.2.pdf⟩ July 2005.

[19] Y. Fu, J. Chase, B. Chun, S. Schwab, A. Vahdat, SHARP: an architecture for secure resource peering, in: Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03), Bolton Landing, NY, 2003.

[20] P. Ganesan, K. Gummadi, H. Garcia-Molina, Cannon in G major: designing DHTs with hierarchical structure, in: Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04), Tokyo, Japan, 2004.

[21] Globus Team, The Globus Project, ⟨http://www.globus.org/⟩ July 2005.

[22] I. Goldberg, D. Wagner, R. Thomas, E.A. Brewer, A secure environment for untrusted helper applications: confining the Wily Hacker, in: Proceedings of the Sixth USENIX Security Symposium, San Jose, CA, 1996, pp. 1–13.

[23] L. Gong, M. Mueller, H. Prafullchandra, R. Schemers, Going beyond the sandbox: an overview of the new security architecture in the Java development kit 1.2, in: Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS'97), Monterey, CA, 1997.

[24] Y.C. Hu, S.M. Das, H. Pucha, Exploiting the synergy between peer-to-peer and mobile ad hoc networks, in: Proceedings of the Ninth Workshop on Hot Topics in Operating Systems (HotOS IX), Lihue, Hawaii, 2003.

[25] A. Iamnitchi, I. Foster, D.C. Nurmi, A peer-to-peer approach to resource location in grid environments, in: Proceedings of the 11th Symposium on High Performance Distributed Computing, Avon Books, New York, 2002.

[26] IBM Corporation, IBM Load Leveler: User's Guide, Technical Report SH26-7226_00, IBM Corporation, 1993.

[27] S. Kenny, B. Coghlan, Towards a grid-wide intrusion detection system, in: Proceedings of the European Grid Conference (EGC2005), Amsterdam, The Netherlands, 2005.

[28] G.H. Kim, E.H. Spafford, The design and implementation of Tripwire: a file system integrity checker, in: Proceedings of the Second ACM Conference on Computer and Communications Security (CCS'94), Fairfax, VA, 1994.

[29] M.J. Litzkow, M. Livny, M.W. Mutka, Condor—a hunter of idle workstations, in: Proceedings of the Eighth International Conference on Distributed Computing Systems (ICDCS 1988), San Jose, CA, 1988, pp. 104–111.

[30] M. Litzkow, M. Solomon, Supporting checkpointing and process migration outside the UNIX kernel, in: USENIX Conference Proceedings, San Francisco, CA, 1992, pp. 283–290.

[31] M. Litzkow, T. Tannenbaum, J. Basney, M. Livny, Checkpoint and migration of UNIX processes in the condor distributed processing system, Technical Report 1346, Computer Sciences Department, University of Wisconsin, Madison, WI, 1997.

[32] R. Mahajan, M. Castro, A. Rowstron, Controlling the cost of reliability in peer-to-peer overlays, in: Proceedings of the Second International Workshop on Peer-to-Peer Systems (IPTPS'03), Berkeley, CA, 2003.

[33] B.P. Miller, M. Christodorescu, R. Iverson, T. Kosar, A. Mirgorodskii, F. Popovici, Playing inside the black box: using dynamic instrumentation to create security holes, Parallel Process. Lett. 11 (2–3) (2001) 267–280.

[34] Napster, Napster file sharing tools, ⟨http://www.napster.com/⟩ July 2005.

[35] Pastry Project Team, FreePastry, ⟨http://freepastry.rice.edu/FreePastry/⟩ July 2005.

[36] L. Peterson, T. Anderson, D. Culler, T. Roscoe, A blueprint for introducing disruptive technology into the internet, in: Proceedings

of the First ACM Workshop on Hot Topics in Networks (HotNets-I), Princeton, NJ, 2002.

[37] R. Raman, M. Livny, M. Solomon, Matchmaking: distributed resource management for high throughput computing, in: Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing (HPDC-7), Chicago, IL, 1998, pp. 140–146.

[38] R. Raman, M. Livny, M. Solomon, Matchmaking: an extensible framework for distributed resource management, Cluster Comput.: J. Networks Software Tools Appl. 2 (2) (1999) 129–138.

[39] R. Raman, M. Livny, M. Solomon, Resource management through multilateral matchmaking, in: Proceedings of the Ninth IEEE Symposium on High Performance Distributed Computing (HPDC-9), Pittsburgh, PA, 2000, pp. 290–291.

[40] S. Ratnasamy, P. Francis, M. Handley, R. Karp, S. Schenker, A scalable content-addressable network, in: Proceedings of the ACM SIGCOMM 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM'01), San Diego, CA, 2001, pp. 161–172.

[41] S. Rhea, D. Geels, T. Roscoe, J. Kubiatowicz, Handling churn in DHT, in: Proceedings of the USENIX Annual Technical Conference (USENIX'04), Boston, MA, 2004.

[42] M. Ripeanu, I. Foster, A decentralized, adaptive, replica location service, in: Proceedings of the 11th Symposium on High Performance Distributed Computing, 2002.

[43] A. Rowstron, P. Druschel, Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility, in: Proceedings of the 18th ACM Symposium on Operating System Principles (SOSP'01), Chateau Lake Louise, Banff, Canada, 2001, pp. 188–201.

[44] A. Rowstron, P. Druschel, Pastry: scalable, decentralized object location, and routing for large-scale peer-to-peer systems, in: Proceedings of the Middleware 2001, IFIP/ACM International Conference on Distributed Systems Platforms, Heidelberg, Germany, 2001, pp. 329–350.

[45] Sharman Networks, Kazaa Media Desktop, ⟨http://www.kazaa.com/⟩ July 2005.

[46] D. Spence, T. Harris, XenoSearch: distributed resource discovery in the XenoServer open platform, in: Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC'03), Seattle, WA, 2003.

[47] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, H. Balakrishnan, Chord: a scalable peer-to-peer lookup service for internet applications, in: Proceedings of the ACM SIGCOMM 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM'01), San Diego, CA, 2001, pp. 149–160.

[48] V. Sunderam, PVM: a framework for parallel distributed computing, Concurrency: Practice Exp. 2 (4) (1990) 315–339.

[49] J.B. Weissman, Gallop: the benefits of wide-area computing for parallel processing, J. Parallel Distrib. Comput. 54 (2) (1998) 183–205.

[50] E. Zegura, K. Calvert, S. Bhattacharjee, How to model an internetwork, in: Proceedings of the IEEE INFOCOM'96—The Conference on Computer Communications, San Francisco, CA, 1996, pp. 594–602.

[51] R. Zhang, Y.C. Hu, Borg: a hybrid protocol for scalable application-level multicast in peer-to-peer networks, in: Proceedings of the 13th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV 2003), Monterey, CA, 2003, pp. 172–179.

[52] B.Y. Zhao, J.D. Kubiatowicz, A.D. Joseph, Tapestry: an infrastructure for fault-resilient wide-area location and routing, Technical Report UCB//CSD-01-1141, University of California, Berkeley, CA, 2001.

[53] S. Zhou, LSF: load sharing in large-scale heterogeneous distributed systems, in: Proceedings of the Workshop on Cluster Computing, Orlando, FL, 1992.

[54] S.Q. Zhuang, B.Y. Zhao, A.D. Joseph, R.H. Katz, J. Kubiatowicz, Bayeux: an architecture for scalable and fault-tolerant wide-area data dissemination, in: Proceedings of the 11th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV 2001), Port Jefferson, NY, 2001, pp. 11–20.

Ali R. Butt received his B.Sc. (Hons.) degree in Electrical Engineering from University of Engineering and Technology Lahore, Pakistan in 2000. He is currently a Ph.D. candidate in Computer Engineering at Purdue University, where he also served as the president of Electrical and Computer Engineering Graduate Student Association for 2003 and 2004. His research interests include distributed resource sharing systems spanning multiple administrative domains, applications of peer-to-peer overlay networking to resource discovery and self-organization, and techniques for ensuring fairness in sharing of such resources. His recent work includes design and implementation of buffer cache management techniques for improving file system performance in modern operating systems. He is a member of USENIX, ACM, and IEEE.

Y. Charlie Hu is an Assistant Professor of Electrical and Computer Engineering and Computer Science at Purdue University. He received his M.S. and M.Phil. degrees from Yale University in 1992 and his Ph.D. degree in Computer Science from Harvard University in 1997. From 1997 to 2001, he was a research scientist at Rice University. Dr. Hu's research interests include operating systems, distributed systems, networking, and parallel computing. He has published more than 50 papers in these areas. Dr. Hu received the NSF CAREER Award in 2003. He served as a TPC vice chair for the 2004 International Conference on Parallel Processing (ICPP-04), and a co-founder and TPC co-chair for first International Workshop on Mobile Peer-to-Peer Computing (MP2P'04). Dr. Hu is a member of USENIX, ACM, and IEEE. For more information about Dr. Hu's current activities, please see http://www.ece.purdue.edu/∼ychu.