A Self-Organizing Flock of Condors

Ali Raza Butt, Rongmei Zhang, and Y. Charlie Hu
Purdue University
West Lafayette, IN 47907
{butta, rongmei, ychu}@purdue.edu

ABSTRACT

Condor provides high throughput computing by leveraging idle-cycles on off-the-shelf desktop machines. 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.

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 [2]. The computational grid [11], popularized by systems such as Globus [13, 9] and Condor [21], 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 [22], Gnutella [12], and Kazaa [28], and formalized by systems such as CAN [25], Chord [29], Pastry [26], and Tapestry [32], 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 [27, 6], application-level multicast [4, 33, 31], and routing in mobile ad-hoc networks [16]. In this work, we present a p2p scheme to facilitate the discovery of remote resources. Although the results in this work are achieved via an innovative marriage of the flocking facility [7] in Condor and the proximity-aware p2p routing substrate offered by Pastry [3], the scheme is applicable to other platforms.

SC '03, November 15-21, 2003, Phoenix, Arizona, USA

Condor [21] provides a mechanism for sharing resources by harnessing the idle-cycles on desktop machines. It enables high throughput computing using off-the-shelf cost-effective components. 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 provide job migration and other features. Under normal conditions, 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 Condor's potential of sharing available resources. First, the central manager is a single point of failure, and in case such a failure occurs, the whole pool becomes unusable. Second, 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.

Condor addresses the issue of sharing resources among multiple pools by a mechanism referred to as flocking [7]. 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 for discovering resources such as those in [9] 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 issuing 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.
- 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, as well as through large-scale simulations involving 1000 Condor pools.

The rest of the paper is organized as follows. Section 2 gives an overview of Condor and structured p2p overlay networks. 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 some related work. Finally, Section 7 provides concluding remarks.

2. BACKGROUND

The proposed work leverages the idle-cycle sharing facilities of Condor [21].

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 to the manager, it is placed on a queue. The central manager then searches for an appropriate resource in the pool to run the job on. The job waits in the queue until some matching resource becomes available. Condor uses an extensive resource description language/matching technique ClassAds [23, 24] to make this match. It allows users to specify the nature of resources their jobs would require, and then finds a match accordingly. In addition, Condor provides checkpointing facilities [20, 19], 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[7]. 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 machines in another pool *B* to be run on its resources, the central manager of *A* is configured to allow this sharing, and the central manager of *B* is also explicitly configured to use resources available in *A*. The manager of *B* will only send jobs to *A* if the local resources are unavailable or in use. The job scheduling negotiations occur between the two managers, and the negotiated job is executed on the remote resource. It should be observed that this mechanism is static, and requires both pool *A* and pool *B* to be pre-configured for resource sharing.

2.3 Structured p2p overlay networks

Structured p2p overlay networks such as CAN[25], Chord[29], Pastry[26], and Tapestry[32] 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 $O(logN)$ overlay hops and each node only maintains $O(logN)$ 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 [26, 3] 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. A more detailed description of Pastry can be found in [26, 3].

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, from 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 source and destination node only by a small constant value [3].

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 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 bootstrap 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 of the proximity-aware routing table that 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.

Figure 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 combined computing power. The Condor pools that are interested in sharing resources with other pools form a p2p overlay network, and each pool is issued a random node identifier (`nodeId`) in the ring. The `nodeId` is randomly assigned, and the nodes that are adjacent in the node identifier space may be far apart in the physical network and vice versa. For instance, the grayed nodes in Figure 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.

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 the pool that is making this determination as the local pool in the following discussion.

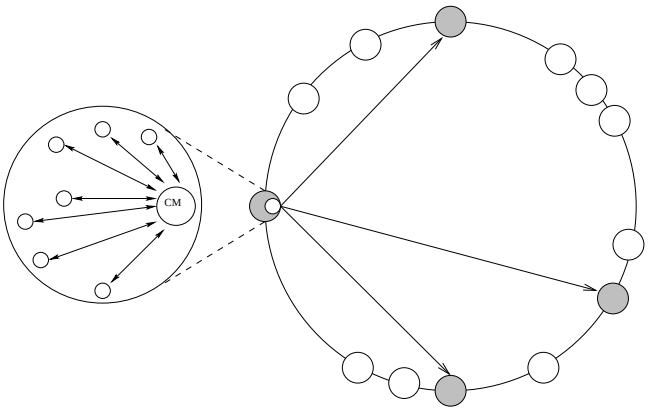

Figure 1: Interactions among Condor central managers of different pools. The big circle on the right represents the managers arranged in a p2p ring (i.e. the circular node identifier space). Each circle on the ring represents a Condor pool. An enlarged version of a pool is shown on the left. The resources 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.

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 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 the p2p layer 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 free and 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 the 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, Figure 1 shows the local pool utilizing resources from various grayed nodes, which are chosen as the Condor pools that are nearby 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 in-

formation about the available resources in its pool, and its desire to share the resources with M. 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 ith sublist containing M_Rs from the ith row of the routing table. Hence, because of 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. Announcements from pools that are unwilling to share their resources are excluded from the `willing_list`. 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. Also, this increases the chance of the needy pools to fairly share the free resources among themselves, making it unlikely to have the first needy pool to reach an available pool taking up all the resources in that pool.

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 will not be able to flock to other resources that do not appear in the Pastry routing table.

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 decrements the TTL, and if the TTL is greater than zero, forwards it 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 contacts them to determine how far they are, and use this information to generate the `willing_list` that is sorted with respect to proximity. Each sublist in this `willing_list` contains nodes which may be several hops away, but 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 separation of the resource discovery mechanisms from the sharing policies of individual pools, hence, giving pool owners full control of how their resources are utilized. 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 [23, 24]. 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 compares metrics such as queue lengths, average pool utilization, and the number of resources available, and based on these comparisons sorts the available pools in order from most suitable to least suitable. This dynamically ordered list is then utilized by Condor to select a 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. The direct matchmaking techniques can also be extended to support matching of local jobs from one pool to resources in remote pools. We plan to study these extensions in our future research.

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 manger 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 the rings. The central manager periodically informs everyone 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. 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, except for correcting the problems with the failed central manager.

3.4 Security

Sharing resources pose security challenges which if not addressed can lead to the compromise of the shared resources. Condor employs authentication of users as well as resources and provides for security policy specification [5]. In a single pool, Condor can be set up to run jobs only from the users who have standard accounts on the resources. Anonymous users and users from remote pools can be executed as user `nobody`, hence curtailing the capabilities of malicious users. In case of self-organized flocking, the jobs from remote pools can also be sandboxed using either the Java Virtual Machine [15] or system-call tracing as proposed in [14, 2], giving a resource fine-grained control over the actions of the job. To protect against a malicious remote condor pool, the proposed approach uses a policy file, which controls 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 authentication 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 security features of systems such as Globus [9, 10, 8, 1] can also be leveraged in the proposed design to ensure more secure operations.

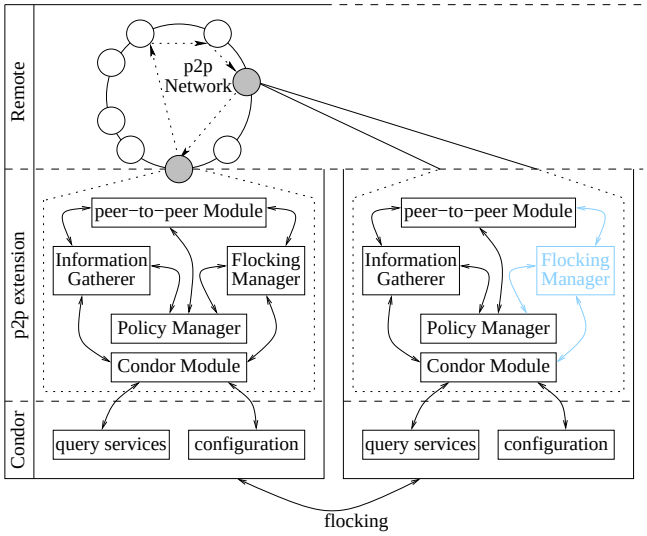

Figure 2: Architecture of *poolD* and interactions of various modules. *poolD* runs on the central manager in each pool that intends to share remote resources, and provides for the self-organization of the flock. The local *poolD* (shown on the left) does not directly interacts with the `Flocking Manager` on the remote *poolD* (shown on the right).

4. IMPLEMENTATION

We implemented the proposed scheme by adding a software layer on top of Condor. The software is designed using the Pastry API [26] (available in Java), and utilizes the flexible configuration control of Condor to dynamically modify the flocking behavior of Condor.

The main software is divided into two components: *poolD* which runs only on the central managers to maintain the self-organized flock and to discover remote Condor pools, and *faultD* which runs on all the resources in a Condor pool to provide resilience to central manager failures.

4.1 *poolD*

Figure 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 utilizing of remote Condor resources is desired. The `peer-to-peer Module` takes care of the p2p routing, the proximity aware routing table, and other bookkeeping tasks related to maintaining up-to-date information about the overlay network. 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 which provide information are called the remote nodes.

The `Information Gatherer` handles the resource availability announcements to inform nearby nodes, and also updates 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* contacts its `Condor Module` to obtain the status of the pool. Next, the `Information Gatherer` consults its `Policy Manager` – a module for implementing pool sharing preferences – to determine what resources can be shared with 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/domain names, and/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 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 use of the `Policy Manager`, on both *L* and *R*, ensures that individual pools have control over the resources on which their jobs are run and vice versa, without affecting all the pools on the ring. 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 update process, 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 creates a sorted list of Condor pools with resources that can be leveraged. In creating this list, the number of free resources available on them as well as the proximity information are taken into consideration. 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, and 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*

Figure 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 all the resources in the pool. It has dual roles: on the submit and/or compute machines it acts as a passive *Listener*, whereas on the central manager it acts as an active *Manager*. Figure 4 shows the protocol followed for switching between the two roles. 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 same software starts on all the resources and the central manager as a *Listener*. 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 [26] to route messages between the nodes.

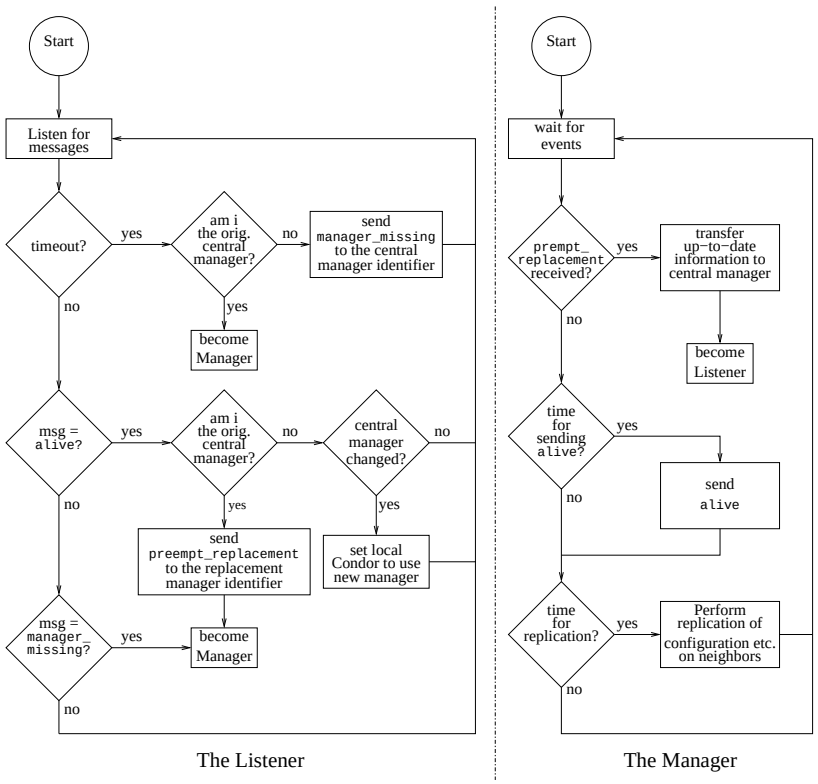

Figure 4: The protocol followed by *faultD* module to switch between the roles of *Listener* and *Manager*.

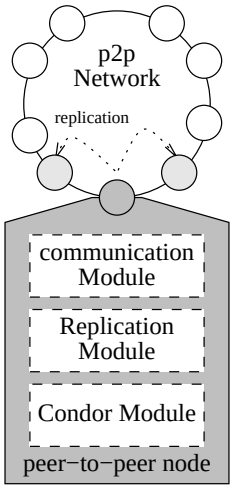

Figure 3: Architecture of *faultD*. It runs on all the resources in a Condor pool, and provides resilience to central manager failures.

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 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* is to periodically broadcast an `alive` message to all the resources in the pool. The message also contains some bookkeeping information that the resources can use to detect a failure at the central manager. When the original central manager is brought online in the presence of an active replacement central manager, the original manager sends 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 stop, the node sends a `manager_missing` message to the previously known `nodeId` of the central manager. 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 the `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. Consequently, the receiving node is the replacement manager. In this event, *faultD* takes on the role of the *Manager*. It already has the replicated pool configurations, and can assume the role of central manager right away.

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 as well as the results from simulating a large number of Condor pools over the Internet.

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 make these measurements, four condor pools were used with three machines each available for computations. The configurations are shown in Figure 5. To observe the effect of various configurations on the scheduling of jobs and the resulting throughput, we first created a synthetic job that would consume resources for any specified amount of time. We then created a sequence of 100 submissions of the synthetic job, each with a random duration between 1 to 17 minutes, issued with a random interval between 1 to 17 minutes, with an average of 9 minutes. We created 12 such job sequences, enough to keep 12 machines busy all the time. For the case of a four separate Candor pools, the 12 job sequences are merged into four different job queues, one for each pool. A job queue with `n` job sequences merged together implies that it on average has `n` job requests issued simultaneously. The number of sequences in the drive queues were 2, 2, 3, 5 for pool A, B, C, and D, respectively. For the case of a single integrated pool, we merged all 12 sequences into a single queue.

In order to use the generated job queues, we implemented a job driver which takes as input the job queues, and submits the specified length synthetic jobs to the respective Condor pools at specified times. The pools were set up so that jobs would never run on the central managers, and would always run on the compute machines. The compute machines were dedicated to these jobs, hence, effects of checkpointing because of an owner returning to the desktop were avoided. The TTL parameter and the expiration interval in availability announcement messages were set to one and 1 minute, respectively. The interval at which the `Flocking Manager` queries the local `Condor Module` in *poolD* was also set to 1 minute.

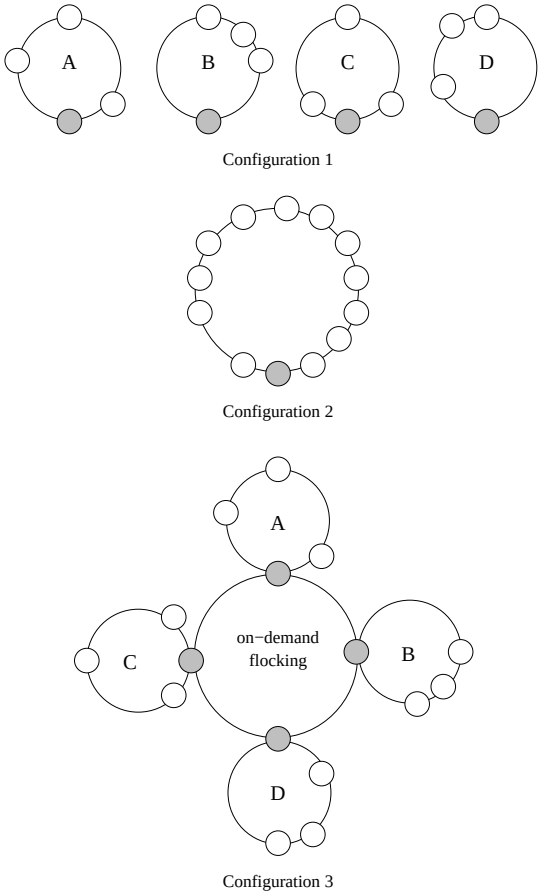

Figure 5: The various configurations of Condor pools used for making performance measurements. Each ring represents a p2p overlay network of nodes. The grayed nodes are the central managers and the white nodes are the compute-machines.

5.1.2 Results

Table 1 shows the wait times of jobs in the queue for the configurations in Figure 5. In Configuration 1, we fed the queues to individual pools without flocking. The number of sequences in the job queues 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 279.48 minutes 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 554.82 minutes.

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 2), and loaded the pool with a queue 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 13.02 minutes. 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, since such merging requires administrative privileges across organizational boundaries [7].

Next, we measured how on-demand flocking can utilize the multi-

Pool	No. of sequences in job queue	Without flocking (Conf. 1)				With flocking (Conf. 3)			
		mean	min	max	stdev	mean	min	max	stdev
A	2	1.01	0.03	14.12	2.32	15.87	0.03	73.12	18.59
B	2	1.86	0.03	18.12	3.19	16.95	0.03	55.70	16.99
C	3	19.18	0.03	63.08	15.86	16.55	0.03	57.78	16.41
D	5	279.48	0.05	554.82	180.15	14.20	0.03	58.92	14.58
Overall	12	121.72	0.03	554.82	177.23	15.52	0.03	73.12	16.19

	No. of sequences in job queue	mean	min	max	stdev
Single Pool (Conf. 2)	12	13.02	0.02	34.70	10.57
Conf. 3 (all load at A)	12	13.06	0.03	34.90	10.63

Table 1: Wait times for jobs in queue. All numbers are in minutes. One job sequence contains 100 jobs of random length of 1 to 17 minutes, issued at random intervals between 1 and 17 minutes.

pool resources with Configuration 3. Here we used the same individual pools of Configuration 1, 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 1. It was observed that compared to 279.48 minutes in Configuration 1, the average wait time for pool D was reduced to 14.20 minutes. In addition, the maximum wait time with flocking is reduced to 10.62% of that without flocking. Pool C has similar performance to that in Configuration 1. The reason for this is that pool C has 3 machines and is driven by a job queue with average issue of 3 jobs at a time. Therefore, pool C does not provide its resources to other pools. The small 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 pool A and pool B is an increase in average queue wait times. This is because pool A and pool 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 may have to wait for the job from pool D to either finish or be suspended and moved. In these experiments, pool A would wait for remote jobs to finish. This is a matter of policy, and the local pool can set up its own preferences. Note, however, that compared to Configuration 1, the overall mean time is reduced by 106.2 minutes, whereas in pools A and B, it is increased by only 14.86 and 15.09 minutes, respectively.

To determine how flocking affects the wait times of jobs when compared to the single integrated pool, we loaded Configuration 3 at one of the pools (A) with the same job queue with 12 sequences as used to load the single pool of Configuration 2. The results show that the wait times in the two scenarios are almost the same. The few seconds difference is due to the fact that in case of flocking, the various Condor managers have to negotiate the available job resources, whereas in single pool, one manager can make the decision.

We further measure the impact of concurrent job loading at multiple pools versus job loading at a single pool on job wait times with flocking (Configuration 3). Table 1 shows that the difference is insignificant. The 2.46 minute difference in the mean times can be explained by the observation that when individual pools are loaded, preference is given to local jobs. 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 potentially lengthens the negotiation process.

In summary, these results show that without requiring resource merging, the self-organizing flocking mechanisms presented in this paper can not only achieve a significant improvement in job throughput over without flocking, they can 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 [30]. The size of the IP network is 1050 routers, 50 of which are used in transit domains and the rest of the 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 in each stub domain, giving us a total of 1000 pools. The Condor central manager in each pool is attached to the domain router by a LAN connection. 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 actual measurements, a synthetic job trace is created to drive the simulations. As before, a single request sequence consists of 100 jobs, issued at random intervals. The delay between any two consecutive job requests follows a uniform distribution between 1 and 17, giving an average delay of 9 time units. The length of each job follows the same uniform distribution between 1 and 17 time units. At each Condor pool, one job queue is created to drive the simulations by merging a random number of such single sequences. The number of single sequences follows a uniform distribution between 25 and 225.

A Condor manager attempts to schedule a job request to the ma-

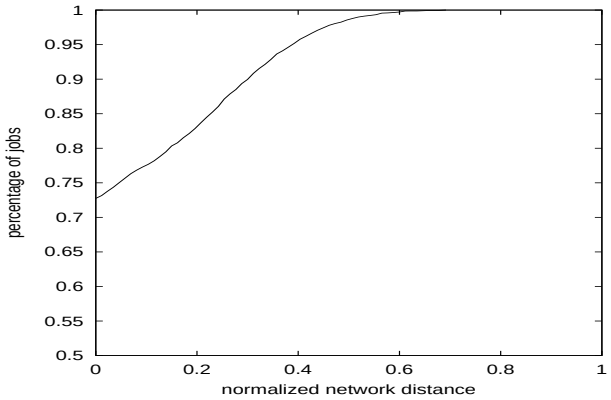

Figure 6: Cumulative distribution of locality for scheduled jobs when flocking is enabled. The x-axis stands for the ratio between the network distance from the job submission pool to the actual execution pool and the diameter of the underlying IP network. Locality zero means that the jobs are scheduled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

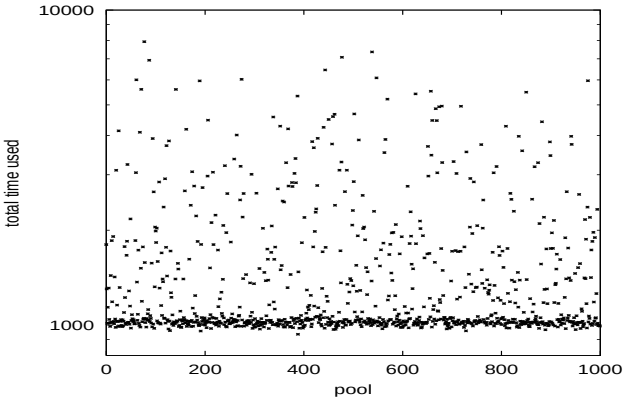

Figure 7: Total completion time at each Condor pool without flocking.

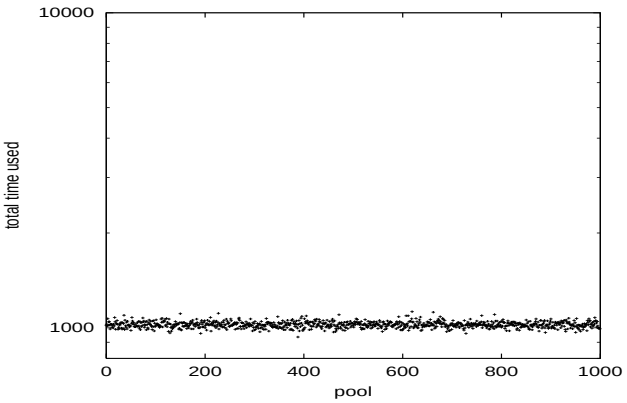

Figure 8: Total completion time at each Condor pool when flocking is enabled.

chines 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 successfully scheduled and the queue at every central manager is emptied. As in the prototype measurements, the TTL parameter and the expiration interval in availability announcement messages were set to one and 1 time unit, respectively, and the interval at which the `Flocking Manager` queries the local `Condor Module` in *poolD* was also set to 1 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. Figure 6 shows the cumulative distribution of the locality for the jobs scheduled by self-organized flocking. The distance measured by the network routing delay between the pool where the job is submitted and the pool where the job is scheduled to execute, 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. Figure 7 shows the total time that it takes to complete all the jobs without flocking, observed at each Condor pool. Similarly, Figure 8 shows the total completion time when self-organizing flocking is enabled. As the figures show, flocking can evenly distribute workloads among all the available resources, hence executing jobs at each Condor pool takes about the same amount of time and all the job queues are emptied almost simultaneously. On the other hand, 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.

In the third set of results, we determine the effects of flocking on the average wait time of a job in the request queue. The wait time in the job queue is the duration between the time unit that a job is submitted and the time unit that the job is dispatched from the queue. Here we measured the average wait time for jobs in all 1000 queues. Figure 9 shows the total time in queue without flocking, and Figure 10 shows the same when self-organized flocking is utilized. The simulation shows that flocking can 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.

6. RELATED WORK

The related work can be divided into two categories: p2p routing facilities, and resource discovery across multiple administrative domains.

Work on p2p mechanisms was initially driven by file sharing programs such as Napster [22], Gnutella [12] and Kazaa [28]. The simple routing protocols of these first generation p2p systems were extended to provide for efficient, scalable and robust routing mech-

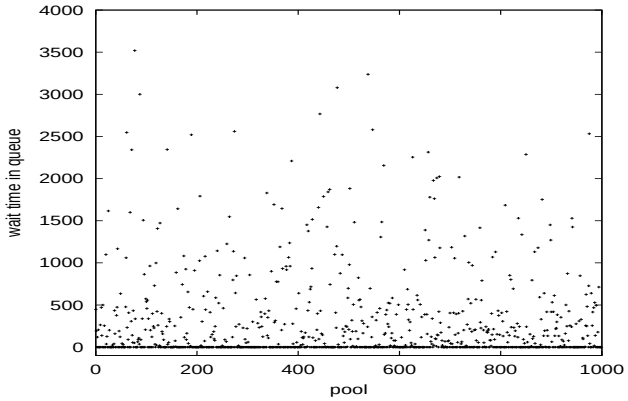

Figure 9: Average wait time in the job queue at each Condor pool without flocking.

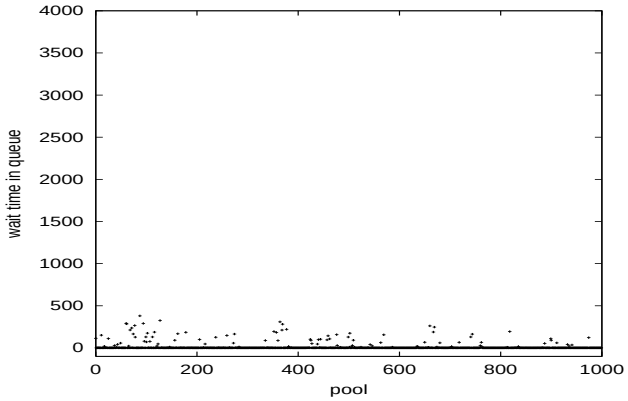

Figure 10: Average wait time in the job queue at each Condor pool when flocking is enabled.

anisms by imposing some form of structure on the overlay network. Examples of structured p2p systems include Pastry [26], CAN [25], Tapestry [32], and Chord [29]. These approaches have many applications ranging from distributed storage facilities [27], scalable group communication systems [4], decentralized p2p web caching [17], as well as enhancing the scalability of multi-hop routing protocols for mobile ad-hoc networks [16]. The focus of this work is to utilize structured p2p systems for resource discovery.

In the area of resource discovery, the approaches adopted in the Grid [11] can be used for sharing resources across administrative domains. Systems such as Globus [9] and PUNCH [18] use a hierarchical system to discover resources, and are centralized in nature. The goal of this work is to develop a simple, robust, and decentralized technique for sharing resources using the p2p technology. It can be extended into the Grid platforms for scalable, distributed resource discovery.

7. CONCLUSION

We have presented a locality-aware p2p approach to remote Condor pool discovery, which yields a self-organizing flock of Condor pools. The static flocking mechanisms that are available in Condor provides 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. Measurements of the approach using four Condors pools with a total of 12 computing machines, driven by a synthetic job trace shows 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. In the future work, we plan to conduct measurements utilizing real job traces, and we anticipate similar results. Simulations using 1000 Condor pools show that locality-aware routing indeed provides bandwidth savings and improve the overall throughput. These results show that p2p technology offers a promising approach to dynamic resource discovery essential to high throughput computing.

Acknowledgment

We thank Miron Livny and the anonymous reviewers for their helpful comments. This work was supported by an NSF CAREER award (ACI-0238379).

8. REFERENCES

[1] R. Butler, D. Engert, I. Foster, C. Kesselman, S. Tuecke, J. Volmer, and V. Welch. A National-Scale Authentication Infrastructure. *IEEE Computer*, 33(12):60–66, Dec. 2000.

[2] A. R. Butt, S. Adabala, N. H. Kapadia, R. J. Figueiredo, and J. A. B. Fortes. Fine-Grain Access Control for Securing Shared Resources in Computational Grids. In *Proc. 16th International Parallel and Distributed Processing Symposium(IPDPS'02)*, Fort Lauderdale, FL, 2002.

[3] M. Castro, P. Druschel, Y. C. Hu, and A. Rowstron. Exploiting Network Proximity in Peer-to-Peer Overlay Networks. Technical Report MSR-TR-2002-82, Microsoft Research, One Microsoft Way, Redmond, WA, 2002.

[4] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. Scribe: A large-scale and decentralised application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications (JSAC) (Special issue on Network Support for Multicast Communications)*, 20(8):100–110, Oct. 2002.

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

[6] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. 18th ACM Symposium on Operating System Priniciples (SOSP'01)*, pages 202–215, Chateau Lake Louise, Banff, Canada, 2001.

[7] 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(1):53–65, May 1996.

[8] I. Foster, N. T. Karonis, C. Kesselman, and S. Tuecke. Managing security in high-performance distributed computations. *Cluster Computing: The Journal of Networks, Software Tools and Applications*, 1(1):95–107, 1998.

[9] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.

[10] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. A Security Architecture for Computational Grids. In *Proc. 5th ACM Conference on Computer and Communication Security (CCS 98)*, pages 83–92, San Francisco, CA, 1998.

[11] I. Foster (Ed.) and C. Kesselman (Ed.). *The GRID: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, 1999.

[12] J. Frankel and T. Pepper. The Gnutella protocol specification v0.4 (2000). ⟨ http://cs.ecs.baylor.edu/donahoo/classes/4321/GNUTellaProtocolV0.4Rev1.2.pdf ⟩ (29 July 2003).

[13] Globus Team. The Globus Project (28 july 2003). ⟨ http://www.globus.org/ ⟩ (29 July 2003).

[14] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A Secure Environment for Untrusted Helper Applications: Confining the Wily Hacker. In *Proc. 6th USENIX Security Symposium*, pages 1–13, San Jose, CA, 1996.

[15] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers. Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2. In *Proc. USENIX Symposium on Internet Technologies and Systems (USITS'97)*, Monterey, CA, 1997.

[16] Y. C. Hu, S. M. Das, and H. Pucha. Exploiting the Synergy between Peer-to-Peer and Mobile Ad Hoc Networks. In *Proc. 9th Workshop on Hot Topics in Operating Systems (HotOS IX)*, Lihue, Hawaii, 2003.

[17] S. Iyer, A. Rowstron, and P. Druschel. Squirrel: A decentralized peer-to-peer web cache. In *Proc. 21st ACM Symposium on Principles of Distributed Computing (PODC 2002)*, pages 213–222, Monterey, CA, 2002.

[18] N. H. Kapadia and J. A. B. Fortes. PUNCH: An architecture for Web-enabled wide-area network-computing. *Cluster Computing: The Journal of Networks, Software Tools and Applications*, 2(2):153–164, Sep. 1999.

[19] M. Litzkow and M. Solomon. Supporting Checkpointing and Process Migration Outside the UNIX Kernel. In *USENIX Conference Proceedings*, pages 283–290, San Francisco, CA, 1992.

[20] M. Litzkow, T. Tannenbaum, J. Basney, and 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.

[21] M. J. M. J. Litzkow, M. Livny, and M. W. Mutka. Condor - A hunter of idle workstations. In *Proc. 8th International Conference on Distributed Computing Systems (ICDCS 1988)*, pages 104–111, San Jose, CA, 1988.

[22] Napster. Napster file sharing tools (n/a online). ⟨ http://www.napster.com/index.html ⟩ (28 July 2003).

[23] R. Raman, M. Livny, and M. Solomon. Matchmaking: Distributed Resource Management for High Throughput Computing. In *Proc. 7th IEEE International Symposium on High Performance Distributed Computing (HPDC-7)*, pages 140–146, Chicago, IL, 1998.

[24] R. Raman, M. Livny, and M. Solomon. Resource Management through Multilateral Matchmaking. In *Proc. 9th IEEE Symposium on High Performance Distributed Computing (HPDC-9)*, pages 290–291, Pittsburgh, PA, 2000.

[25] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A Scalable Content-Addressable Network. In *Proc. ACM SIGCOMM 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM'01)*, pages 161–172, San Diego, CA, 2001.

[26] A. Rowstron and P. Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer systems. In *Proc. Middleware 2001, IFIP/ACM International Conference on Distributed Systems Platforms*, pages 329–350, Heidelberg, Germany, 2001.

[27] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. 18th ACM Symposium on Operating System Priniciples (SOSP'01)*, pages 188–201, Chateau Lake Louise, Banff, Canada, 2001.

[28] Sharman Networks. Kazaa Media Desktop (25 June 2003). ⟨ http://www.kazaa.com/index.htm ⟩ (28 July 2003).

[29] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proc. ACM SIGCOMM 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM'01)*, pages 149–160, San Diego, CA, 2001.

[30] E. Zegura, K. Calvert, and S. Bhattacharjee. How to Model an Internetwork. In *Proc. IEEE INFOCOM'96 - The Conference on Computer Communications*, pages 594–602, San Francisco, CA, 1996.

[31] R. Zhang and Y. C. Hu. Borg: A hybrid protocol for scalable application-level multicast in peer-to-peer networks. In *Proc. 13th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV 2003)*, pages 172–179, Monterey, CA, 2003.

[32] B. Y. Zhao, J. D. Kubiatowicz, and 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.

[33] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, and J. Kubiatowicz. Bayeux: An Architecture for Scalable and Fault-tolerant Wide-Area Data Dissemination. In *Proc. 11th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV 2001)*, pages 11–20, Port Jefferson, NY, 2001.