Resource Discovery In Semi-structured P2P Networks for CPU Cycle-Sharing

[Extended Abstract]

João Pedro Gomes Neves
Instituto Superior Técnico
MEIC - Alameda 55384
jpgn@ist.utl.pt

ABSTRACT

Today, there is a number of projects that attempt to take advantage of the excess CPU cycles existing in machines connected to the Internet. However, these projects are mainly focused in providing distributed code execution for large scale research projects and do not make it possible for normal users, which contribute their cycles for those projects, to use the same infrastructure to execute common desktop applications for their own benefit.

In the context of this problem, this dissertation studies and proposes resource discovery mechanisms, as well as Peer-to-Peer network topologies that allow the creation of a distributed resource sharing system. This solution is part of the GINGER project, which aims to offer a resource sharing network capable of interacting with the many existing desktop applications without the need to modify them, using techniques found in institutional Grid networks, resource sharing systems as the ones mentioned above and the popular Peer-to-Peer file sharing networks.

1. INTRODUCTION

Over the last two decades and following Moore's Law, the available computing power throughout the world shifted from large mainframes and supercomputers to the ever increasing number of personal computers connected to the Internet, as these became more powerful at a very fast rate. However, most of the computing power offered by these machines is wasted on idle processes, and as personal computers grow more powerful this trend tends to increase.

To use this untapped resource, several projects like BOINC[1] and SETI@Home[2] emerged, giving research institutes the ability to perform massive computations without having to acquire or maintain an expensive grid environment. Through these projects ordinary computers could contribute their unused CPU cycles to scientific research by downloading work units from each project's servers, perform the necessary calculations and uploading the obtained results.

Although these projects approach their desired goals, they fail to meaningfully give back to their contributors, as people who contribute with CPU cycles for these projects are unable to use those same resources for their own benefit.

The approach followed by BOINC and SETI@Home has some disadvantages, however, including the fact it requires users (in this case, researchers) to be able to compile custom versions of the applications whose work they wish to distribute and as such is not an option for most end-users. Additionally, as previously mentioned, the computing power supplied by the contributors is still in the hands of a very small number of institutions, and users cannot tap those same resources to accomplish tasks that matter to them, as ordinary PC users.

On GINGER[11], the authors propose a peer-to-peer based infrastructure for CPU cycle sharing based on the concept of Gridlets, which allow for the use of unmodified applications on the data to be processed, and as such enables every user not only to contribute by helping other users on their computations but also allows them to submit their own work to the environment, in a mutualistic manner. Such an approach requires each node in the network to be able to locate the required resources for the work it needs to do in an efficient and scalable manner, but empowers each user with the ability to use the available cycles to perform everyday tasks which would usually require very large amounts of computing power or time.

The resource discovery problem in institutional Grid infrastructures and distributed cycle sharing projects such as the ones mentioned above is relatively well known and several centralized solutions have been proposed. On a fully decentralized peer-to-peer architecture, however, this problem is further amplified by the lack of global knowledge of the nodes composing the network, and a scalable solution for the general case is difficult to achieve.

Beyond the issue of discovering resources like CPU and disk storage, it is also necessary to tackle the problem of discovering services, which may be provided by a software application present in a system, or physically through the machine itself (e.g. a network printer).

1.1 Contributions

This work introduces a resource discovery mechanism for P2P networks based on the Pastry overlay, more specifically in the context of the GINGER platform. It aims to evaluate several resource discovery mechanisms to be used on the GINGER platform.

A resource in this system can be any of the following: CPU power and availability, memory capacity, disk capacity, network bandwidth, installed applications, services, peripherals, users and others that may apply.

The chosen mechanism should ideally be:

1. **Scalable:** The overall efficiency should not decrease significantly with a larger network;
2. **Complete:** If the resource to be discovered exists, it shall be discovered;
3. **Efficient:** The overhead caused by the discovery protocol should not disrupt the normal functioning of the network;
4. **Flexible:** We aim to support multi-attribute ranged queries, as it is necessary on such a heterogeneous environment.

2. RELATED WORK

Several articles and projects have been produced on the topic of resource and service discovery on institutional grids, cycle-sharing desktop grids and peer-to-peer networks. In this section, some of this work is presented and discussed.

We will focus on a few aspects of each system that are relevant for their study in general and worthy of note in the scope of this project. The most important aspects pertain to overall system architecture and resource discovery.

All these types of infrastructure have their own merits and demerits and have been subject to extensive study in order to provide clues on how they can be combined to take advantage of the best features each system has to offer[10].

2.1 Institutional Grids

Grid computing is used to describe a technique that utilizes the resources of several computers simultaneously in order to solve complex problems which typically require very large amounts of computation power due to either the nature of the problem or the sheer volume of data that needs to be analyzed.

Institutional grid networks have been the main form of distributed computing platforms available worldwide, mainly to large research or academic centers.

Grid network environments are usually centrally controlled by a single entity, which makes these environments implicitly trusted, and as such, less vulnerable to malicious use. This allows for a simpler or non-existent result validation procedure.

2.2 Desktop Grids

Cycle-sharing desktop grid networks are built using regular desktop computers, whose idle CPU cycles are often voluntarily donated by regular computer users. These cycles are often used to perform distributed computations on very large data sets for research purposes.

BOINC, with over half a million active computers providing an average of over 1400 TeraFLOPS of computing as of the time of this writing, is a clear proof that users are willing to provide their idle cycles and that there is an enormous amount of processing yet to be exploited worldwide. The emergence of the use of GPUs for massively parallel processing is expected to greatly increase the total throughput of this kind of grid.

2.3 Peer-to-Peer Networks

Peer-to-peer (P2P) networks have become one of the driving forces of widespread online content production and consumption worldwide. Unlike traditional client-server networks, P2P networks enable cooperation between users without the existence of centralized, third-party authorities and effectively removing the need for costly infrastructure as well as eliminating the single point of failure present in traditional systems.

In a P2P network, as its name implies, every client can also act as a server without requiring any additional configuration or special setup. Peer coordination in a P2P network is essentially free and can be achieved in an autonomous manner, without the need for special peers in the network, although in some cases it is desirable to have a number of peers empowered with extra capabilities for the purposes of coordination, commonly known as super-peers or ultrapeers.

The decentralized and distributed nature of P2P networks makes them highly resilient to the common issues affecting traditional centralized systems. In particular, P2P nodes are highly resilient to Denial-of-Service attacks, distributed or otherwise, and typically enjoy increased efficiency with a greater number of peers present in the system at any given time.

Some peer-to-peer networks organize their nodes according to a given topology, usually distinct from the underlying physical topology, in order to better achieve their goals in terms of robustness, performance and scalability.

A type of peer-to-peer network designed to solve this problem can be a Distributed Hash Table or DHT, where nodes are responsible for a subset of all the keys in the network's key space and cooperate to provide scalable and efficient key location.

Well known DHTs include Pastry[6] and Chord[9], which organize the nodes in a logical ring and route requests through other nodes in the ring.

P2P networks on the whole comprise a large part of all Internet traffic and as such have proved to be scalable, with room for future growth.

3. ARCHITECTURE

One of the goals of GINGER as a platform is to obtain the benefits of Grid-like computing infrastructures with the flexibility of P2P networks in order to enable its use by the masses. For this, it is important to determine a good P2P overlay topology, as the resource discovery mechanisms depend on this topology, as opposed to the underlying physical networks.

The main goal should be a system that obtains the best nodes available for a given work unit based on a number of criteria such as available bandwidth, processing power and so on.

3.1 System Architecture

This project is based on the GINGER platform, shown in Figure 1.

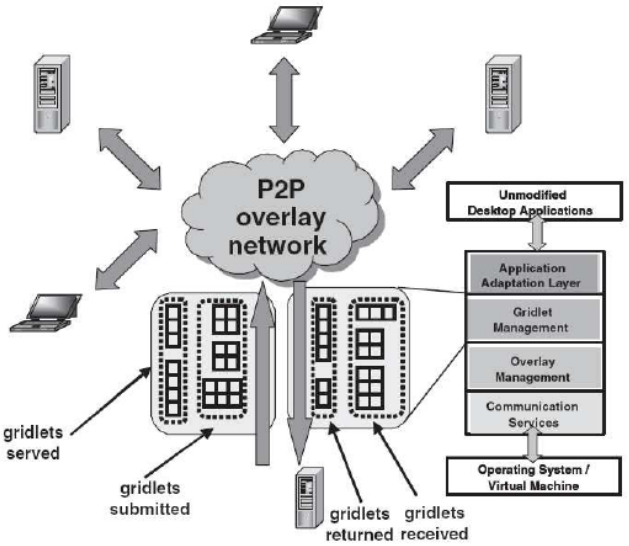

Figure 1: Ginger architecture

GINGER is a middleware platform running on top of a semi-structured P2P overlay network whose basic work unit is a *Gridlet*. A Gridlet contains information regarding the data workload as well as the transformations required to be applied on it. Each Gridlet also contains the *cost* associated with performing those transformations to the data and optionally the actual code that performs them.

The Gridlet application model is divided into three stages: Gridlet creation, Gridlet processing and Gridlet-result aggregation. In gPastry, no Gridlet processing is done at this level, as it is irrelevant for the problem at hand. Also, Gridlets are simplified and only contain information regarding the required resources (application and other requirements) for the Gridlet to be processed.

Whenever a node needs to submit work to the network, the input data is chunked into one or more Gridlets that will then be routed throughout the P2P overlay for processing. When the necessary transformations are computed, the results are packaged in special result Gridlets, which can either be sent directly to the origin or become stored throughout the overlay for later retrieval using a distributed storage system such as PAST[7] or, alternatively, a distributed caching system as the one described in Squirrel[4].

As shown in the above figure, GINGER is composed of four layers:

Application Adaptation layer. The Application Adaptation layer is responsible for the interaction with the unmodified desktop applications, in order to launch them and feed the data received from gridlets, as well as collecting the resulting data from the transformations performed on the original data by the application.

Gridlet Management layer. On the Gridlet Management layer, files and data are partitioned into the corresponding gridlets and received gridlet-results are reassembled into result files as those that the application would produce if ran locally on a single machine.

Overlay Management layer. The Overlay Management layer is, as its name implies, responsible for maintaining the overlay network in order to exchange gridlets with other nodes. This layer only maintains the logical structure that forms a node's information about that overlay however, and all actual network data transfer and communication is handled by the Communication Services layer.

This layer holds all the information regarding the overlay and transparently performs resource discovery in order to locate the resources required by the upper layers. Both super-peers and regular peers maintain a common subset of information, including known super- peers and their own capacity, with super-peers maintaining extra information regarding the availability of all applications its child peers have registered with it, as well as the aggregate capacities announced by other super-peers regarding the applications which fall under the application family that super-peer is responsible for.

Communication Services layer. As previously mentioned, the Communication Services layer's function is simply one of performing data transfer throughout the network. It performs no routing or other type of computation in order to determine an incoming or outgoing message's destination and leaves that function to the Overlay Management layer.

The major focus of this work is on the *Overlay Management* layer, as it is the layer responsible for maintaining the overlay and performing discovery, as explained above.

3.2 Overlay network topology

The overlay network used in gPastry is based upon the Pastry[6] overlay, as it presents a robust P2P overlay, while using a number of modifications to accommodate our resource discovery protocol, which will be detailed across the next sections.

The immediate concerns when designing the architecture for this work were to both minimize the number of messages that need to be exchanged between nodes in the network, to avoid flooding, as well as attempt to provide enough coverage such that the required resources will be found when they are needed, if they are present in the network.

In order to accomplish this, nodes in the network are divided into two sets: regular nodes and super-peer nodes. This closely follows Gnutella's approach using "ultrapeers" for data file indexing, as well as CCOF's *rendezvous points*.

Super-peer nodes perform every function a regular node does and, additionally, they maintain and share high-level information regarding other nodes in the network. As such, super-peers are not distinct from regular nodes but perform additional tasks with regards to overlay management when compared with the remaining nodes in the overlay.

The overall layout of the nodes is displayed in Figure 2. For reference, note that the overlay network does not need to connect nodes in any particular order for super-peers to exist. Repeated links between nodes are omitted in this figure. Let us explore super-peers in more detail in the following section.

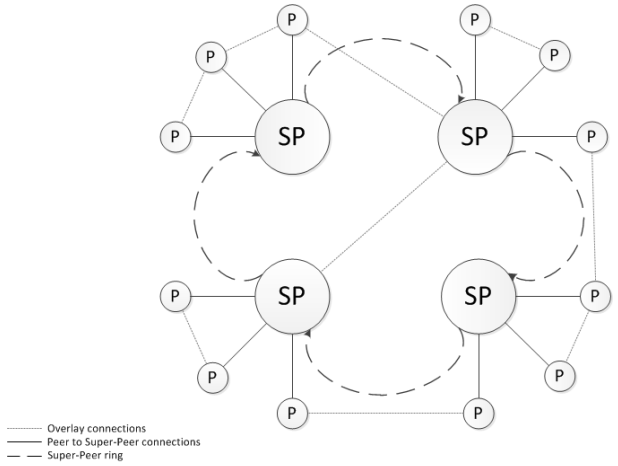

Figure 2: Node organization in the network. "SP" indicates a super- peer, while "P" indicates a regular peer.

Super-peers. Super-peers form a ring amongst themselves to allow faster communication and avoid hopping messages through the overlay network. They share information about the availability of applications among their child nodes and act as resource brokers, sharing their child nodes' resources with each other when such is needed (for instance, when a super-peer's children cannot perform the work required for a given gridlet, it requests work from another super-peer which whose children have enough availability to perform the task).

Regular nodes, in turn, are "clustered" around these super-peers and use them to perform service discovery and gridlet requests to other nodes in the network. Every node is assigned to a single super-peer, determined from all known super-peers through the Pastry proximity metric it is then referred to as that node's *primary super- peer*. The use of the proximity metric was chosen because it explicitly represents the degree of proximity between two given nodes in the overlay, as defined by the basic Pastry protocol.

Despite being assigned to a single super-peer, every node possesses a list of existing super-peers in order to choose a new super-peer in case of failure of the existing one.

Super-peers election is performed in a way that attempts to balance the ratio of super-peers to regular nodes and make sure no nodes are *orphaned*, i.e. left without knowing any super-peers. Nodes will periodically check the number of super-peers they know; if this number is below a certain threshold, those nodes can select themselves to become super-peers with a given probability. This probability has to be kept low enough so that the number of super-peers cannot grow too large at any given point in time, but such that allows a fairly rapid expansion of the number of super-peers in case of shortage.

3.3 Resource Discovery

Applications are the main resource taken into consideration in gPastry and the resource discovery mechanism focuses mostly on finding nodes which have the given application installed.

In order to structure information regarding applications throughout the overlay, using the topology described in the previous section, two pieces of information are kept about each application: the *family hash* and the *application hash*.

The *family hash* is used to describe the family or class of applications any given application belongs to. In the context of gPastry, the family hash is a hash of the *canonical name* of an application, which in this case is, for instance, the URL of the application's main web page online (e.g. for the Python application, the canonical name would be "`www.python.org`"). This *family hash* is used to aggregate information regarding all nodes with an application belonging to that family in the same super- peer, in a way that will be described further in this section.

The *application hash*, on the other hand, describes a specific combination of application name and version and is used to determine, in a given super-peer containing information about a given application family, which nodes possess a specific application installed in the system. For instance, considering the Python application family example, there could be a number of distinct application versions described uniquely by this hash such as `Python 2.6` and `Python 3.0`.

This distinction using the application's version is important for dealing with cases where different versions of a given application are incompatible with each other and thus cannot be used to perform work on the same set of data, or the user requires functionality present only in a specific (or more recent) version. Note that although only the version number is used in the given examples, different "versions" can be easily created for applications running in different operating

systems as well.

As described previously, there are a number of super-peer nodes whose purpose is to aggregate information regarding the installed applications. These super-peers are "responsible" for one or more application families, which means they aggregate the high-level information on all current super-peers regarding those application families.

In particular, each of these super-peers maintains a table containing other super-peers' aggregate availability for a given application inside the family they are responsible for as well as the availability of every child node (including themselves) for each application those nodes register when joining the network, even if that application falls outside the family it owns. However, regarding the capacity available under other super- peers for those families, each super-peer only maintains the aggregate available capacity, with no information about what specific nodes under that super-peer possess the application or their individual capacity. Furthermore, super-peers do not store such information about applications whose family is not of their responsibility when this information does not belong to one of its direct child nodes.

Responsibility is determined according to the application's *family hash*, which is mapped into the super-peers' set of keys. If a super-peer's node ID is larger (according to the Pastry proximity metric) than the key given by the family hash, which belongs to the same key space, then the super-peer is responsible for that family. This is done in order to evenly distribute the application family's key space among all super-peers and avoid overloading any given one. Whenever a new super-peer joins the ring, every super-peer adds it to its super-peer lists and if the new super-peer should become the one responsible for an already existing application family, the super-peer containing that information sends it to the joining super-peer. Super-peers can be responsible for multiple families, as long as the family hash is lower than their own node ID but not lower than any other super-peers' node ID.

When submitting a Gridlet, nodes send it to their super-peer who, according to the information aggregated from his child nodes and other super-peers, distributes the workload of that given Gridlet in the following manner:

- The super-peer forwards the Gridlet to all its child nodes which have the required application and capacity;
- If more nodes are needed, the super-peer contacts the super-peer deemed responsible for that application family by forwarding the Gridlet. Otherwise the Gridlet is dispatched and the super-peer does nothing else;
- Upon receiving the request from another super-peer, the super- peer responsible for that application family checks the capacity table for another super-peer possessing enough child nodes to fulfill the request and forwards the Gridlet to it.
- If no other super-peers exist that can fulfill the request, the responsible super-peer simply replies immediately with a message.

Node capacity is determined by a number of different factors, such as CPU power, total memory and bandwidth. When a node's capacity changes (due to work in progress or external factors, such as a local user interacting with the system), its new capacity is sent to its corresponding super-peer.

Upon receiving this information, the super-peer updates its internal aggregated capacity for the applications which this particular node provides and, if the new aggregate capacity is different enough (configurable for increments of, for instance, 10, 100, etc.) from the previous value it had transmitted to the super-peer responsible for that application, it sends an update message to that super-peer in order to update the value that super-peer is keeping.

The use of this type of asynchronous, on-demand communication instead of either having synchronous updates or sending updates every time a node updates its availability, is done in order to prevent flooding the network with messages regarding relatively insignificant changes to the overall availability of a given application.

Additionally, upon joining the network a node sends all the information regarding its installed applications to its newly determined super-peer, which then updates its aggregate capacity for each application registered by the new node and if the change in capacity for a given application is significant enough it then updates the super-peer responsible for that application, as in the previous situation.

3.4 Data Structures

gPastry builds on top of Pastry overlay and as such, a number of data structures used in Pastry are found unmodified in this work. These data structures include the basic node state information in the form of three separate structures: **routing table**, **leaf set** and **neighborhood set**. Additionally, a number of new data structures were developed to support super-peer information as well as available resource information.

To complement and enhance Pastry in order to support the desirable new features for gPastry, a number of data structures were developed to hold and organize new information.

The most relevant data structures pertain to storing information about existing super-peers, as well as resources and their availability. These structures are described in the following sections.

Super-peer information. Nodes must maintain some information about existing super-peers in order to advertise their resources and become available for remote computation. To do so, nodes store a sorted circular list of known, live super-peers. This list is centered on the closest super-peer according to the Pastry proximity metric and is sorted by proximity to the current node. This particular super-peer is designed as the *primary* super-peer for that node and is used as the main point of communication between the node and the remaining overlay for purposes of resource discovery.

Whenever a new super-peer is discovered, or one joins or leaves the network, this list is updated accordingly. The

closest super-peer is designated as the *primary* super-peer and is used as the primary choice for resource advertisement and lookup functionality. This allows to minimize the number of hops necessary to communicate with a super-peer to obtain information, thus preserving Pastry's performance characteristics. A super-peer's primary super-peer is always itself per the proximity metric. This introduces no inconsistencies, and does not preclude super-peers from submitting Gridlets themselves; they will follow the exact same protocol as other peers.

Resource availability information. Resource availability information is maintained only by super-peers and functions as a distributed index where super-peers can lookup resources both inside and outside their node group with different levels of detail. Super-peers maintain resource availability for each node under their respective groups, namely which applications are available among the many nodes, the actual nodes that possess them and their current availability. This allows the super-peers to pick the most available nodes for a given Gridlet.

In order to locate resources outside of their group, however, super- peers maintain only the aggregate availability of other super-peers' groups for the applications they advertise amongst themselves. This allows not only to locate more resources for a known application outside a super-peer's group in case of saturation, but also allows to discover nodes which can provide different resources which are do not exist in the current group. In order to minimize the amount of information each super-peer has to maintain, super-peers only exchange this information with the super-peers responsible for the relevant application families.

For instance, if a super-peer A gains or loses capacity related to an application X in a given family F, only the super-peer responsible for family F is informed of those changes and records the appropriate information. Other super- peers are not informed as the information is not relevant for the application families they are responsible for.

4. IMPLEMENTATION

For the implementation of gPastry, the chosen tools were Java 6 SE and the PeerSim[5] P2P network simulation platform version 1.0.3. Since PeerSim only provides a protocol-independent simulation platform, the entire code base that implements the Pastry protocol was written from the ground up for the purposes of this project.

Using this platform as opposed to use others such as FreePastry[3], which already implements Pastry, allows greater flexibility if the need to change the underlying protocol would arise, at the cost of extra development effort. PeerSim is developed for the Java platform, and all development for it is done in Java as well.

4.1 The PeerSim platform

PeerSim has two basic types of simulator: an event-driven simulator, where progress is done by acting on events triggered by reception of messages sent from one node to the next, and a cycle-driven simulator, which allows to act on the simulation at intervals.

Both of these simulators can be integrated seamlessly to take advantage of the benefits inherent to each type of simulator. For the purpose of this work, this was the followed approach, using the event- driven simulator for most of the simulation as well as attempting to increase the realism of the simulation, and using the cycle-driven simulator to allow for easier execution of periodic functions on the nodes.

For this work, a few controls have been developed. They allow for the manipulation of the simulated network in run time and are important for the evaluation of gPastry.

PeerSim allows for the control of communications on the simulated overlay at the transport level through classes in the `peersim.transport` package. For the purpose of this work, the `UniformRandomTransport` was the one selected, which implements a transport layer which reliably delivers messages with a random delay, drawn from the interval defined in the configuration file using a uniform distribution. This simulates an environment where packet losses do not occur and/or can be corrected by retransmitting packets with delay (typical in real world applications where TCP is usually favored in detriment of UDP as the underlying transport protocol).

Additionally, we can select a *wiring* structure for the overlay. This defines a low-level topology which emulates the way nodes would be physically connected to each other in a real-world situation. PeerSim allows for the selection of several wiring methods (implemented through PeerSim controls that are executed upon initialization). The Internet is, at its core, a semi-structured network where most nodes are interconnected indirectly through other nodes, with a variable number of hops between them. As such, the control selected to wire the nodes in the network was the `peersim.dynamics.WireKOut` control, which randomly connects nodes among each other. The out degree k of the nodes can be passed as a parameter to this control through the PeerSim configuration file.

4.2 GINGER implementation

The GINGER platform was partially implemented as part of gPastry, with some of the layers, like the Application Adaptation layer, being implemented only as stubs, for they were not relevant for the purpose of this work. The implemented layers will be described in a bottom-up approach.

Communication Service. The Communication Service layer, abstracts the actual communication with other nodes in the overlay. Most of the implementation details are further abstracted into the `PastryProtocol` class, for ease of implementation within the PeerSim platform.

This class provides the following methods: `route`, which routes the message through the overlay; `routeMyMsgDirect`, which sends a message directly to a given node, bypassing the routing mechanism; `deliver`, which processes an incoming message at a higher level than the one provided by the `PastryProtocol` class.

The routing methods wrap the corresponding methods present

in the `PastryProtocol` class, which provide the required functionality. This is because of implementation details imposed by the PeerSim platform which are not relevant for the work at hand.

Overlay Management. The Overlay Management layer is implemented in the `OverlayManager` class and performs the essential functions for maintaining overlay information, as well as performing resource discovery through the protocols developed for gPastry.

This layer contains, per node, the relevant information required to route messages to other nodes, as well as information regarding existing super-peers. Message routing is performed using the Pastry routing mechanism and as such this layer must maintain the relevant data structures, namely the routing table, leaf set and neighborhood set inherited from Pastry.

In order to locate resources according to the protocol we developed, the Overlay Management layer must maintain additional data structures, described later.

Gridlet Management. Gridlet creation happens at this level, with the data received from the Application Adaptation layer, described previously. Upon Gridlet reception on the Overlay Management layer, it is forwarded through to the Gridlet Management layer, where it is processed. If the node currently has enough capacity to process the Gridlet, it is passed on to the Application Adaptation layer. If not, it can be forwarded to a different node.

Gridlets arriving for processing are forwarded to the Application Adaptation layer which will later return the data required for the creation of a Gridlet Result containing the result of transforming the Gridlet data on through the required application, which is cached for later forwarding to the initiating node.

Application Adaptation. The Application Adaptation layer was implemented only as a stub, playing no role in this work, as its purpose is to seamlessly integrate with the applications needed for Gridlet data processing.

This layer passes the Gridlet's data to the application after performing whatever transformations are needed on that data in order for the application to work correctly and receives the resulting output data. This output data should then be passed on to the Gridlet Management layer, where a new result Gridlet is created. For the purposes of this project, we can assume the Gridlet result is a simple transposition of the input data, such as an arithmetic operation or hash code.

4.3 gPastry implementation

For gPastry, Pastry was implemented according to the original Pastry paper[6] and as mentioned previously is realized in the `PastryProtocol` class. Messages arriving on a node are processed through the `processEvent` method, which dispatches messages to the `route` method for the message to be actually routed (or delivered in this node, according to the Pastry routing algorithm) if the node has already joined the network.

If the message is to be delivered to the current node, it is then dispatched to the `deliver` method, which will forward it to the `CommunicationService` class, which implements GINGER's Communication Service layer.

A node joins the network through the `join` method. An overloaded method is provided for the first node to join the network in order to simplify the process. Whenever a node sends a message to another, the `route` message dispatches that message using the Pastry routing protocol. The underlying mechanism for message transmission is provided by PeerSim through the `send` method, which in this case is wrapped in another method with the same name to simplify development.

Messages. Messages used in this implementation of Pastry form the basis of all messages exchanged in the network, from gPastry's point of view.
The base class is `RawMessage`, which defines the base methods and fields all messages should share.

Due to PeerSim constraints, messages are sent as Java objects. Nonetheless, these objects could easily be transposed into a generic, language-independent format for real-world use.

Upon joining the network, a node A sends a `PastryJoinRequest` message whose destination node is also A to its bootstrap node B. B then routes the message through the network using the original Pastry protocol. Every node through which the message passes through will reply to A with a `PastryJoinReply` message containing the corresponding routing table node, as per the original Pastry initialization protocol so that A can build its initial state.

Gridlet messages add a number of extra fields to accommodate for the additional information they must carry. These fields contain the application hash, the application family hash as derived from the provided canonical name, its required availability and any input data that should be processed by the application at the destination node. As with other messages, the Gridlet message is implemented as a Java object for the purposes of this specific implementation, but can easily be translated into more appropriate formats for real-world use.

The basic format for a Gridlet is shown in Figure 3.

4.4 Super-peer implementation

Pastry does not contemplate different types of nodes and as such is not completely adequate for the proposed architecture. On top of the base Pastry overlay, a super-class of nodes was added to perform the work of super-peers.

Super-peers are, as previously mentioned, a subset of all nodes in the network. As such, they share a common code base in the form of the `PastryProtocol` class.

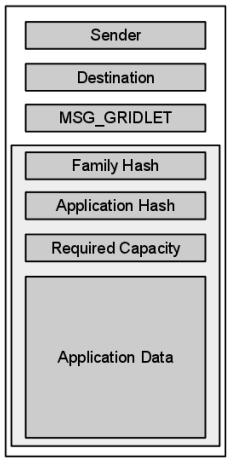

Figure 3: Structure of a Gridlet.

In order to implement super-peers, nodes already possess fields for handling the additional information. This approach was chosen instead of simply using the Java language inheritance mechanism due to issues found with the PeerSim platform, which, at the time this work was being implemented, did not properly support subclasses as PeerSim protocol implementations.

With this approach, what distinguishes regular nodes from super-peer nodes is the `isSuperPeer` boolean flag and the presence of an instance of the `SuperPeerInfo` class in the `spInfo` field inside a node, which encapsulates all information the super-peer possesses.

This class contains the following mappings, from which a super-peer can obtain the information necessary to determine where sufficient nodes exist to process the currently received Gridlet:

- A mapping from application hashes to a map containing all known nodes with that application installed and their current capacity;
- A mapping of all currently known super-peers to their respective aggregate capacities.

When a Gridlet G is received by a super-peer X from one of its child-nodes and reaches the overlay management layer, it is processed through the `gridletReceived` method in the following manner: The super-peer determines which super-peer is responsible for the application family to which the Gridlet refers to.

In the case X is the super-peer responsible for that given application family, it attempts to obtain child-nodes which can perform work on G and forwards it to at most three of those children.

Replicating work on multiple children decreases the chance for a Gridlet never to be worked upon due to the node processing it leaving the network at any time and can also yield better performance than the simpler case of assigning the Gridlet to a single node in the case the chosen node would suddenly have reduced capacity due to external intervention (e.g. a user begins locally using the node for intensive applications). This follows the work of Silva et al on *Workqueues with Replication* (WQR)[8]. Note that if a given super-peer has an application whose family hash is the one that super-peer is responsible for, the super-peer is a child node of itself.

There can be the case where there are no children nodes fit to perform work on the Gridlet X received. In that case, X looks up the other super-peers it is aware of and their respective aggregate capacity for the required application.

If X is not responsible for G, then it determines the super-peer Y by looking up in the remaining known super- peer lists and forwards the Gridlet to Y, which repeats this process until the Gridlet is forwarded to the correct child nodes and processed.

4.5 Resource representation

Existing applications are described through simple XML files. These contain information describing the application, such as its name, canonical name, version and executable location (unused in this work).

These descriptions are read and parsed into a specific internal data structure. For the purpose of this work, the `location` field is not considered, but could be used by GINGER's *Application Adaptation* layer to launch the required application. This information is local to each node and super-peers are not required to store all this information, and instead only store the hashes obtained from this description, which are generated by the peers who wish to register this application.

The use of XML for the application description allows for a greater flexibility on the level of detail by which each application is described. If an application's minimum runtime requirements are known beforehand, these could also be included in specific elements.

5. EVALUATION

In order to evaluate gPastry, a number of objective metrics were defined that will allow to determine the performance of the system under different conditions.

As the nature of this work relies on gathering data on a sufficiently large peer-to-peer environment which is not possible to achieve in real-world conditions without an already existing, widespread implementation of the system, all metrics were measured through the use of the PeerSim simulator and its network observation capabilities.

In a peer-to-peer environment, there are a number of important metrics that allow us to evaluate its overall performance and establish comparisons between different systems.

First and foremost, one must evaluate the load the protocol has on the network. This can be done by measuring the amount of messages traded between nodes as part of the protocol. This allows us to have an idea of the impact the gPastry protocol has in the overall network performance, as

well as determine which parts can be improved to avoid congestion and flooding.

Secondly, it is useful to measure the amount of messages that actually reach their destination within a certain number of hops. This allows to measure the overall efficiency of the overlay. Measuring this, as well as the ratio of requests that do not find an appropriate node, is of interest in determining the resource discovery protocol's efficiency in various scenarios, such as one with resource scarcity or excess.

Additionally, we measured the average number of hops required to discover a resource through the sending of a Gridlet, as well as the average number of hops spent in super-peer redirections in order for the Gridlet to reach an appropriate node.

It would be of interest to compare the results with other algorithms and protocols such as random-walk, but this could not be done in time for this dissertation due to the need of implementing those protocols.

The work done on gPastry was evaluated by using the PeerSim P2P network simulator. PeerSim allows for the creation of "controls", which can act as entities external to the entire network, thus providing them with global knowledge about the network and it's constituent parts.

Since the tests are being processed in a simulated environment, all time units refer to the number of simulation cycles processed. For this work, we use a number of 500000 cycles. In each cycle, nodes can send or messages with low probability. However, the total number of messages will still be extremely large due to the duration of the simulation.

In effect, in each case, the key points to test were:

- Effectiveness - Whether the desired resources were located in a limited number of hops;
- Efficiency - Number of messages necessary to locate the desired resources, as well as number of hops required for a Gridlet to reach the appropriate destination;

Effectiveness. In order to evaluate the effectiveness of the protocol we analyze the ratio between the total number of Gridlets that were sent by the peers and the number of those Gridlets that actually reached a node where they were processed at various points in the simulation.

In this simulation through Figure 4 we see that initially, when nodes are still joining the network, a large number of Gridlets do not find any appropriate node in the allotted number of hops. However, as more nodes join the network and register their availability in super-peers, the amount of Gridlets that successfully locate the required resources increases, stabilizing at different points in the various cases. Efficiency stabilizes more rapidly with larger numbers of nodes, as expected since there is a greater number of super-peers and a larger number of peers from which resources can be tapped. The number of super-peers existing in the network grow slowly, in the end accounting for less than 1% of the total number of peers.

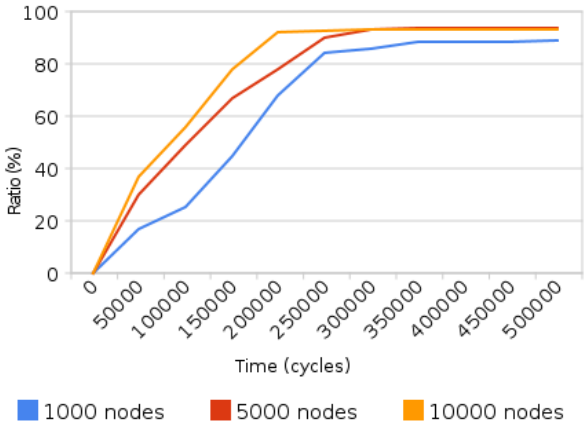

Figure 4: Resource discovery effectiveness

N	Super-peers	Gridlets	Successful
1000	27	125498	111693
5000	63	637298	598677
10000	109	1374089	1277902

Table 1: Effectiveness statistics

Efficiency. In order to evaluate gPastry with regards to efficiency, we have to look at the number of messages required to locate resources, as well as the number of hops Gridlets must take throughout the overlay in order to locate them. The results presented next show that, even with the introduction of super-peers, the number of hops necessary for a Gridlet to reach its destination is not significantly impacted.

There is a number of hops spent in super-peer redirection, but it levels rapidly as the number of super-peers stabilizes and they become known amongst themselves as to become nearly insignificant. This is due to the fact that in all cases, the number of super-peers is small enough that they tend to be known among each other fairly rapidly through the use of `PastrySuperQuery` messages. The average amount of hops for each test can be seen in Figure 5. The number of messages exchanged throughout the simulation is shown to be fairly high and there is much space for improvement, especially in regards to the amount of messages spent in querying for super-peers and the subsequent replies. The results are illustrated in Figure 6.

These messages alone account for more than 10% of all traffic in the network, which is overwhelmingly high for messages whose only function is to keep nodes informed of the current super-peers. The basic messages inherited from the Pastry overlay do not significantly impact the protocol, as they account for about 5% of the entire amount in the current tests, a number that should be lesser in a real world environment where the overlay does not come up entirely at the same time.

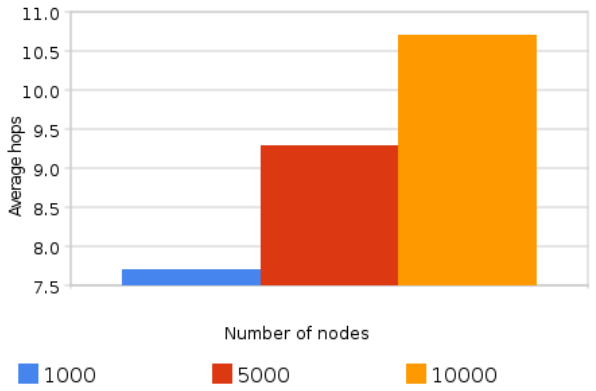

Figure 5: Number of hops for Gridlets

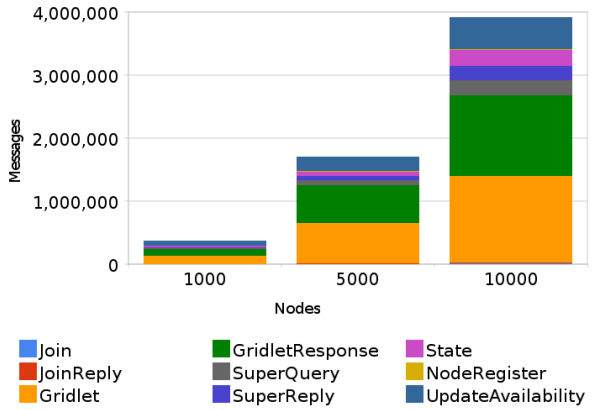

Figure 6: Number of messages exchanged, by type

6. CONCLUSION

In this work, a resource discovery model for Peer-to-Peer networks in the context of CPU cycle sharing was presented. As the available computational capacity available online grows as a consequence of Moore's Law and the expansion of broadband access, new protocols that allow users to exploit these resources in their everyday tasks grow ever more important. Through the use of systems such as GINGER, popular applications can transparently increase their perceived performance by leveraging the massively parallel computing power available throughout the world.

The obtained results show that the approach followed in the development of this system works and that resource discovery in P2P environments can be achieved efficiently through the use of a hierarchical topology with super-peers, an approach that is already well known from existing file-sharing Peer-to-Peer applications such as Gnutella. The amount of messages exchanged using the protocol in this work were clearly sub-optimal, but we believe this is more due to the current implementation, which tends to send far more messages than those required for the purposes of maintenance, than to the actual protocol.

The amount of information to be stored in each peer can become significant after large periods of time but intelligent management of such information, added to the fact that many peers only remain connected for relatively small amounts of time minimize this problem. Overall, the overhead imposed by maintaining the additional information is compensated by the improvements this protocol brings in the area of resource discovery.

7. REFERENCES

[1] D. P. Anderson. Boinc: A system for public-resource computing and storage. In *GRID '04: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, pages 4–10, Washington, DC, USA, 2004. IEEE Computer Society.

[2] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. Seti@home: an experiment in public-resource computing. *Commun. ACM*, 45(11):56–61, 2002.

[3] P. D. et al. FreePastry. `http://www.freepastry.org`.

[4] S. Iyer, A. I. T. Rowstron, and P. Druschel. Squirrel: a decentralized peer-to-peer web cache. In *PODC*, pages 213–222, 2002.

[5] M. Jelasity, A. Montresor, G. P. Jesi, and S. Voulgaris. The Peersim simulator. `http://peersim.sf.net`.

[6] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Lecture Notes in Computer Science*, pages 329–350, 2001.

[7] A. I. T. Rowstron and P. Druschel. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. In *SOSP*, pages 188–201, 2001.

[8] D. P. D. Silva, W. Cirne, F. V. Brasileiro, and C. Grande. Trading cycles for information: Using replication to schedule bag-of-tasks applications on computational grids. In *Applications on Computational Grids, in Proc of Euro-Par 2003*, pages 169–180, 2003.

[9] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *IEEE Transactions on Networking*, 11, February 2003.

[10] D. Talia and P. Trunfio. Toward a synergy between p2p and grids. *IEEE Internet Computing*, 7:96, 94–95, 2003.

[11] L. Veiga, R. Rodrigues, and P. Ferreira. Gigi: An ocean of gridlets on a "grid-for-the-masses". *Cluster Computing and the Grid, 2007. CCGRID 2007. Seventh IEEE International Symposium on*, pages 783–788, May 2007.