CORE Metadata, citation and similar papers at core.ac.uk
ded by Texas A&M University

DISPATCH: DISTRIBUTED PEER-TO-PEER SIMULATIONS

A Thesis

by

KUNAL S. PATEL

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

December 2007

Major Subject: Computer Science

DISPATCH: DISTRIBUTED PEER-TO-PEER SIMULATIONS

A Thesis

by

KUNAL S. PATEL

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Approved by:

Chair of Committee,	Dmitri Loguinov
Committee Members,	Donald Friesen
	Narasimha Reddy
Head of Department,	Valerie E. Taylor

December 2007

Major Subject: Computer Science

ABSTRACT

Dispatch: Distributed Peer-to-Peer Simulations. (December 2007)

Kunal S. Patel, B.E., Sardar Patel University

Chair of Advisory Committee: Dr. Dmitri Loguinov

Recently there has been an increasing demand for efficient mechanisms of carrying out computations that exhibit *coarse grained* parallelism. Examples of this class of problems include simulations involving Monte Carlo methods, computations where numerous, similar but independent, tasks are performed to solve a large problem or any solution which relies on ensemble averages where a simulation is run under a variety of initial conditions which are then combined to form the result. With the ever increasing complexity of such applications, large amounts of computational power are required over a long period of time. Economic constraints entail deploying specialized hardware to satisfy this ever increasing computing power.

We address this issue in Dispatch, a peer-to-peer framework for sharing computational power. In contrast to grid computing and other institution-based CPU sharing systems, Dispatch targets an open environment, one that is accessible to all the users and does not require any sort of membership or accounts, i.e. any machine connected to the Internet can be the part of framework. Dispatch allows dynamic and decentralized organization of these computational resources. It empowers users to utilize heterogeneous computational resources spread across geographic and administrative boundaries to run their tasks in parallel.

As a first step, we address a number of challenging issues involved in designing such distributed systems. Some of these issues are forming a decentralized and scalable network of computational resources, finding sufficient number of idle CPUs in the network for participants, allocating simulation tasks in an optimal manner so as to reduce the computation time, allowing new participants to join the system and run their task irrespective of their geographical location and facilitating users to interact with their tasks (pausing, resuming, stopping) in real time and implementing security features for preventing malicious users from compromising the network and remote machines.

As a second step, we evaluate the performance of Dispatch on a large-scale network consisting of $10 - 130$ machines. For one particular simulation, we were able to achieve up to 1500 million iterations per second as compared to 10 million iterations per second on one machine. We also test Dispatch over a wide-area network where it is deployed on machines that are geographically apart and belong to different domains.

To my parents who have always been my inspiration

ACKNOWLEDGMENTS

I sincerely thank my research advisor, Dr. Dmitri Loguinov, for his valuable insights and regular discussions on various issues pertaining to this work. I am also thankful to Dr. Donald Friesen and Dr. A.L. Narasimha Reddy for serving on my committee. I appreciate the help and company of all the members of the Internet Research Lab and friends in College Station. I am especially thankful to Xiaoming Wang and Derek Leonard for their Latex and formatting tips and to Yueping Zhang for providing me valuable tips on technical writing. Lastly, I am grateful to my sister for supporting and encouraging me throughout my work.

TABLE OF CONTENTS

CHAPTER Page

LIST OF FIGURES

CHAPTER I

INTRODUCTION

A. Motivation

Parallel computation has been an essential component of scientific computing for decades. Traditionally, when one thinks of parallelization, one often envisions *fine-grained* parallelization, which requires substantial inter-node communication utilizing protocols such as MPI [12] or PVM [10]. Recently, however, there is an increasing demand for efficient mechanisms of carrying out computations that exhibit *coarse-grained* parallelism. This class of problems include computations where numerous similar but independent tasks are performed to solve a large problem or any solution that relies on ensemble averages where a simulation is run under a variety of initial conditions which are then combined to form the result, several graph theory related problems where simulation needs to run for long time to obtain accurate results, problems related to analyzing stock market behavior etc. In such applications, all tasks are independent of each other. Each machine runs the task as an independent entity. Some popular examples of these kind of applications are Seti@Home [21] project, where data from astronomical measurements is farmed out to many PCs for processing, and when completed returned to a centralized server and post processed, simulations involving *monte carlo* methods where large number of iterations (typically infinite) are required to obtain desired accuracy. With the ever increasing complexity of such applications, it is necessary to use all the available computing resources. Economic constraints entail deploying specialized hardware to satisfy these ever increasing computing power.

The journal model is *IEEE Transactions on Automatic Control*.

Grid Computing has tried to answer this challenging problem. But due to its complexity and restrictions, it is not widely used and is only studied in research academia. Moreover, Grid Computing projects focus on class of problems that are very different from those discussed above. Grid Computing has several different definitions based on how it is utilized by its users. According to [14], a simple, serviceable definition for the concept of Grid Computing would be

Definition 1. *Grid Computing allows users to unite pools of servers, storage systems, and networks into a single large system that can deliver the power of multiple-systems resources to a single user point for a specific purpose. To a user, data file, or an application, the system appears to be a single enormous virtual computing system.*

Scope of such systems is limited to research institutes or business organizations. Centralized administration is responsible for authentication, task scheduling, fault detection, for monitoring the system and other book keeping. User needs to be the part of organization to leverage the computational power provided by such systems. Centralized administration restricts the scalability of systems and as a result such systems are not widely deployed and used.

Also, Grid Computing projects are aimed at applications that require high performance and availability for a short period of time. Such systems are called High Performance Computing (HPC) environment. HPC environments are measured in terms of FLoating point Operations Per Second (FLOPS). In contrast to that we are targeting applications that require large number of computing resources for long period of times. Users of such applications are interested in knowing how many tasks they can complete over a period of time instead of how fast an individual task can complete. For instance, in Seti@Home [21] project, scientists are interested in processing as many chunks of data as possible over a period of time without caring about how long each chunk of data takes for processing.

During the past two decades, the Internet has seen an explosive growth. With the increasing advancement in technology, number of connected devices are constantly growing. These devices often remain connected and idle for large chunks of time for e.g., at night times, at lunch time, at meetings etc. The challenge is to harness this available unused computing resources spread across the globe for running large and complex parallel applications.

Peer-to-peer networks have recently gained lot of attention in the social, academic and commercial communities. Several file sharing systems such as gnutella [11], bittorrent [1], Kazaa [15] are publicly available. From a research perspective systems such as CAN [19], Chord [22], Pastry [20] and Tapestry [24] have demonstrated the ability to serve as a robust, fault-tolerant and scalable substrate for a variety of applications. Distributed storage facilities built on these protocols include CFS [5], OceanStore [16] and PAST [7]. Some other examples of peer-to-peer applications include routing in mobile ad-hoc networks [13], application level multi cast [3, 23] etc.

B. Our Contribution

In this work we design and implement a decentralized and scalable peer-to-peer framework where *anybody* connected to the Internet can leverage the idle resources of all the participating machines to run their applications in parallel. In contrast to grid computing and other institution-based cycle sharing systems [17], Dispatch targets an open environment, one that is accessible to all the users and which do not require any sort of membership. It does not require registration or accounts at web sites i.e., any machine connected to the Internet can participate as “service provider” or "service consumer" or both at the same time. Dispatch is designed to ensure that only *idle* CPU cycles are utilized. If a user is logged on a participating machine and is doing some interactive work then it will not be utilized. It does not force participants to provide their CPU for remote tasks. Participants can act as "resource provider" or "resource consumer" voluntarily and are always allowed to withdraw their membership at any point of time.

We design two distinct architectures based on target deployment. We call them *S-Dispatch* and *U-Dispatch*. S-Dispatch is based on structured peer-to-peer network substrate called *Chord* [22]. It scales to millions of machines but is relatively complex and has a large overhead for implementing several protocols. U-Dispatch is based on unstructured peer-to-peer networks. It is targeted for environment where system needs to scale for *only* few hundred machines. Typical examples of such environment are research lab, business organizations etc. It is relatively simpler than S-Dispatch and does not have a large overhead for its several functionalities but is restricted in scalability.

Both framework makes sure that user is able to join the network and utilize the available resources irrespective of geographic location. *Node join* protocol is both scalable and decentralized. Each machine joins the overlay network by connecting few existing peers in the network. These peers act as *neighbors* of the node joining the framework. The protocol ensures that the single machine is not overloaded with neighbors and at the same time average degree of each machine (number of neighbors) do not exceed a small constant value in U-Dispatch and $O(logN)$ in S-Dispatch where N is the number of machines in the framework. Our periodic *neighbor exchange* protocol ensures that the machine always remain connected to the network and thereby improving the resilience of network and making probability of node isolation very small.

Dispatch task allocation design ensures that task is allocated in an optimal manner on all available resources so as to reduce the computation time of the task. S-Dispatch achieves load balancing in probabilistic sense while U-Dispatch runs a deterministic algorithm for optimal task allocation . It allows user to decide the number of resources needed for running their task. If resources required are more than the available resources then participant will able to add more resources as they become available. Idle machine detection and allocating task to them is a dynamic process. User do not need to submit their task to some centralized system as in Grid Computing. Also it does not require resource reservation for future use. It allows user to run their task on all available resources immediately.

Writing simulation in Dispatch is a simple process and is different than most of the available systems. In Dispatch, same application acts as frontend for task submitter and as simulation process for remote machines. Simulation processes are started as background tasks under *idle priority class.* This is what makes it different from the concept of *CPU cycle stealing.* If a local process is utilizing the CPU then the remote tasks will not be able to utilize the CPU. Scheduler in Operating System will never schedule the remote task as it runs with lowest priority. On the other hand, if CPU is idle then all the tasks started on that machine will get equal CPU cycles since all of them run with the same priority i.e., idle priority. Dispatch allows real time interactions with the simulation tasks running on remote machines. User can pause, resume and stop any simulation task running on remote machine. Similarly, local user, whose resource is utilized, can also stop or pause and resume all the tasks running on the machine. Dispatch also allows access to intermediate results of the simulation. This is achieved by the *send update* method.

Our security model exploits API's provided by Windows Operating System. It does not require any extra software or libraries. Since remote user is not aware about the identity of task submitter, such systems require strong security policies. We ensure that the malicious user is not able to modify the data of remote machine with the help of OS built in features called *user tokens* and *impersonation*. It does not allow remote users to read, write or modify any data on remote machines. Only resource accessible to the user is idle CPU cycles. We achieve CPU cycle sharing system in true sense. User do not need to share anything else with other participants. If Dispatch is deployed on machines belonging to same domain then we make sure that job submitter belongs to the domain of resource using NT LAN Manager (NTLM) protocol, current security protocol in Windows for authenticating remote users.

C. Thesis Organization

Chapter II gives a brief description on peer-to-peer systems and the related work done in this field. Chapter III talks about S-Dispatch framework along with its components and their functionality. Chapter IV talks about U-Dispatch and its protocols. We implement and check the performance of U-Dispatch under different scenarios. Chapter V describes the implementation of U-Dispatch and the process of writing simulations along with one sample simulation. We describe the experiments performed to check its performance in Chapter VI and Chapter VII concludes the thesis and discusses the future work.

CHAPTER II

BACKGROUND AND RELATED WORK

This Chapter talks briefly about peer-to-peer networks and the related research done in this field.

A. Peer-to-Peer Networks

Peer-to-Peer networks came into prominence with the inception of Napster [18] in 1999. Since then lot of research work is done to understand the various properties of such large distributed networks. The basic idea is to form an overlay network between peers irrespective of their geographic location. They differ from the traditional client/server model. In such models, all the communication takes place between a centralized server which act as resource provider and client which act as resource consumer. Client machines do not interact with each other directly. Such models suffer from lack of scalability. Servers become a single point of failure for the system. On the other hand, in peer-to-peer networks, all the peers are considered equal. Communication directly takes place between them. Peers connect to each other using ad-hoc connections. Figure 1 and 2 depicts this differences. Peer-to-peer networks can broadly be divided into two categories 1) Structured peer-to-peer networks 2) Unstructured peer-to-peer networks

1. Structured Peer-to-Peer Networks

Structured peer-to-peer networks are based on Distributed Hash Tables (DHT). DHTs provide a lookup service similar to a hash table: (key, value) pairs are stored in the DHT, and any participating node can efficiently retrieve the value associated with a given key. All participating machines are assigned a unique uniform random number,

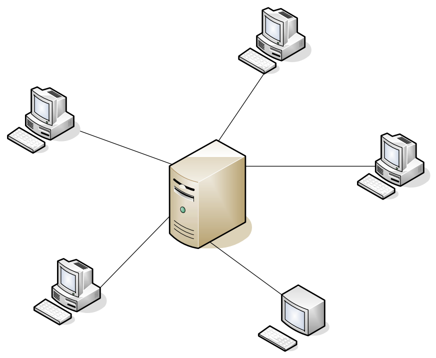

Fig. 1. A typical client-server model

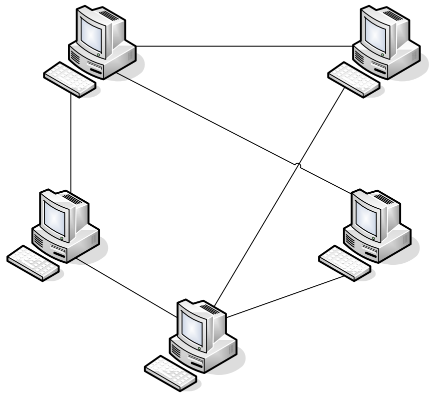

Fig. 2. A typical peer-to-peer model

nodeId, from a large *id space* (e.g., 128 bit unsigned integer). Application specific objects are assigned keys, selected from the same id space. Each key is mapped by the overlay to a unique live node, called the keys root. Mapping is done based on some pre-defined rule. Different systems use different mapping functions. The routing protocol routes messages with a given key to its associated root. To route messages efficiently, each node maintains a routing table with nodeIds of other nodes and their associated IP addresses. Moreover, each node maintains a neighbor set, consisting of some number of nodes with nodeIds near the current node in the id

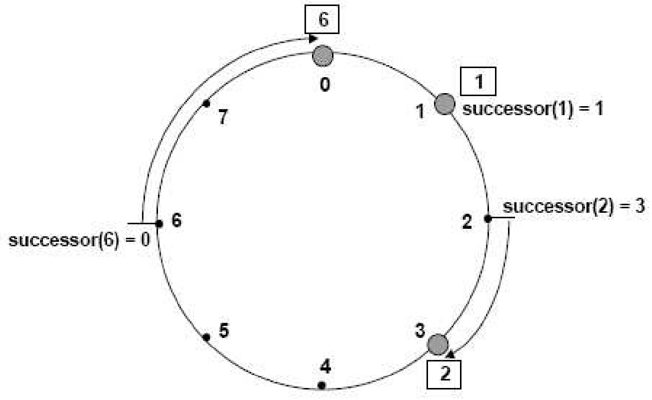

Fig. 3. Identifier circle with m = 3

space. Such highly structured peer-to-peer designs are quite prevalent in research literature but are completely absent in current networks. Examples of such networks include CAN [19], Chord [22], Pastry [20], and Tapestry [24]. Moreover, it is not clear how well such designs work with an extremely transient population of nodes, an inherent characteristics of today's peer-to-peer networks.

2. Consistent Hashing

S-Dispatch is based on structured peer-to-peer substrate called Chord which is based on consistent hashing. So next we discuss consistent hashing in brief. Assume that nodes are located on an identifier circle. They store the data in form of (key, value). Nodes as well as keys are assigned a unique m bit id from the same identifier space. Each node is responsible for storing few keys and associated data. Keys are assigned to nodes in following manner: Key k is assigned to the first node whose identifier is equal to or follows the identifier of k in the DHT space. This node is called the *successor* of key k. Figure 3 shows the identifier circle with $m = 3$. It is taken from [22]. It has three nodes namely node 0, node 1 and node 3. node 0 is responsible for keys $4, 5, 6, 7$ and 0. node 1 is responsible for key id 1 and node 3 is responsible for key id 2 and 3. Each node also maintains a list of other nodes available in the system for routing the search queries. Selection of these nodes (neighbors) is based on rule defined in [22]. Nodes search data by looking up associated key. If they do not have any information on the key, they forward the search query to neighboring node. Process continues until the key and associated data are discovered.

3. Unstructured Peer-to-Peer Networks

Unstructured peer-to-peer networks are much simpler than structured networks. There is neither a centralized file system like client/server model nor they have any precise control over network topology or file placement. Network is formed by nodes joining the system based on some loosely defined rules as in [11]. Unlike structured peer-to-peer networks, the placement of files is not based on any knowledge of the topology. To find a file, a node queries its neighbors. The most typical query method is flooding, where the query is propagated to all neighbors within a certain TTL (Time to Leave) value. These unstructured designs are extremely resilient to nodes entering and leaving the system and relatively easy to deploy. Examples of such system include Bit-torrent [1], Kazaa [15], Gnutella [11] etc.

B. Related Work

This section talks about the related work done in this field.

1. OurGrid

OurGrid [4] closely resembles to the proposed work. OurGrid is a semi centralized system that arranges machine belonging to different domains in a peer-to-peer network. Each domain has a dedicated web site which keeps track of all the resources

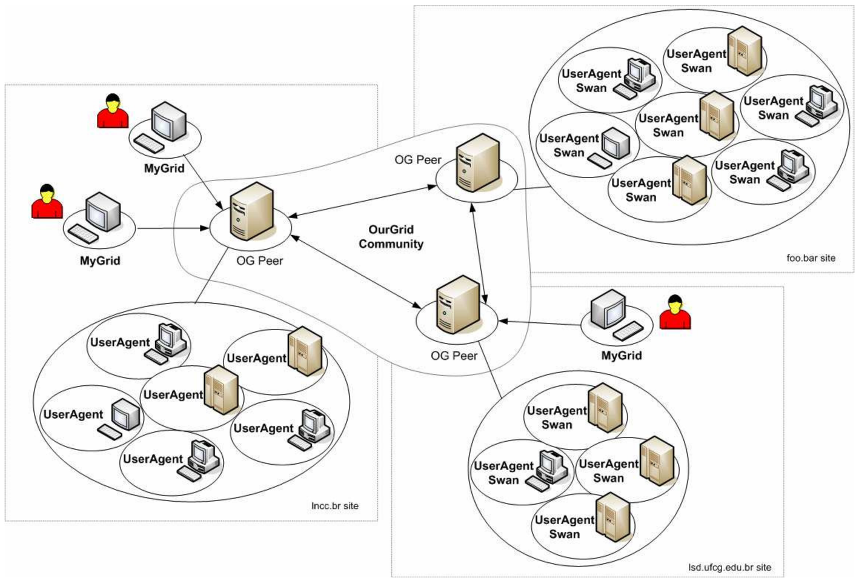

Fig. 4. OurGrid framework

belonging to that domain. User who wants to run the simulation will submit the job using MyGrid, client side software, to OG Peer (centralized authentication and task scheduling system). OG Peer will schedule the job as required resources become free. To use OurGrid, each user needs to create accounts at domain web sites. OG Peer will contact only those domains where the user has an account. The scope of such project is limited to research institutes. Our work goes one step ahead. Any body connected to the Internet can participate in the network either by playing the role of "resource provider" or "resource consumer". See Figure 4.

2. Condor

Condor [17] is another CPU sharing system. In Condor, there is a centralized *match maker*, machine responsible for planning and scheduling of pending task. All the resource provider and agents (people who wants to utilize condor) advertise themselves to their local match maker. Match maker will match the idle resource with the pending job waiting for execution based on policies set by both agent as well as resource provider. From the agent's point of view, policies may include number of machines, type of job, execution time etc. and from the resource point of view it may include time it is going to be idle, allowed users etc. Match maker will match these policies and inform both agent and resource provider about it. After matching, agent can directly run the job on remote resource. If sufficient number of resources are not found under local match-maker then remote match makers are contacted and resources are made available to the agent. Condor also provides facility for check points and task migration if the remote user starts interactive work. There are two constraints that limit Condor's potential of sharing available resources. First, the central match maker is a single point of failure, and in case such a failure occurs, the whole pool (cluster of machines) becomes unusable. Second, the size of individual pools is limited by the resources available to an organization. Also, Condor is a batch system and does not provide environment for real time execution. It is possible that all the resources are running some jobs and so the new agent has to wait till the resources become free.

3. Xtrem Web

Xtrem Web [9] is another such project. All the users who wants to participate have to register at XtremWeb website. It's architecture consist of *worker* and *server* machines. Worker machines work as resources provider while the server machines act as task scheduler. User simulation are submitted to the server machines directly. A idle worker machine will advertise itself to the server machine with a work request and server will respond by scheduling the pending simulation on it.

Server machines are responsible for maintaining the current state of network. During the computation the worker periodically invokes *workAlive* to signal its activity to the server. When a worker machine does not signal its activity for a long time,

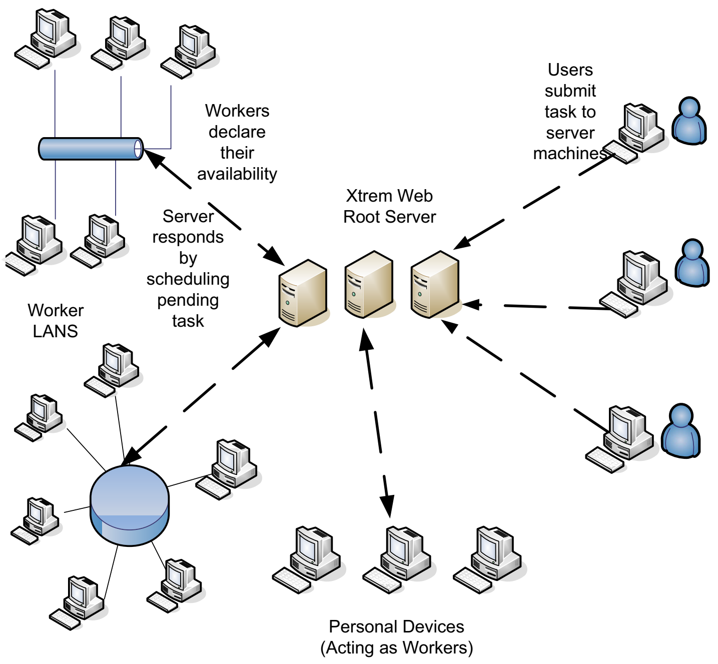
Users submit task to server machines
Workers declare their availability
Xtrem Web Root Server
Server responds by scheduling pending task
Worker LANS
Personal Devices (Acting as Workers)

Fig. 5. Overview of Xtrem Web global computing framework

it is considered to be dead and its task is reschedule to another machine. At the end of computation the worker sends back results to the specified address, through the *workResult* call. The server echoes this message to the job submitter so that the user can collect the results back. See Figure 5.

4. Project JXTA and P3

JXTA defines a set of protocols for developing peer-to-peer applications. Peers in JXTA network self-organize themselves into *peer groups*. A peer group represents an ad hoc set of peers that have a common set of interest and have agreed upon a common set of policies (membership, routing, searching etc). Communication in such systems is achieved by *advertisements*. Advertisements are language-neutral meta data structures represented as XML documents. Peers cache, publish, and exchange

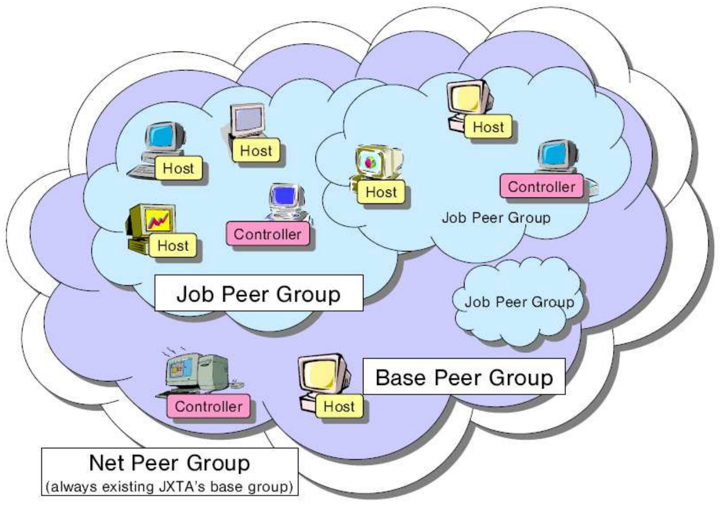

Fig. 6. Organization of job management software and related peer groups in P3

advertisements to discover and locate available resources.

P3 project is based on the JXTA protocols. In P3, user who wants to run the task forms its own task group and publishes it to the network. Resource providers discover the job group, determine whether they want to contribute to the job, and join the group if they do. If no user is present on the machine, then decision is made based on the policies set by the user. P3 relies on multi cast for communication purpose and resource discovery. See Figure 6.

5. Other Projects

Other projects like SETI@home [21], distributed.net [6], BOINC [2] are successfully implemented and globally used. Even though it utilizes the same idea of harvesting idle CPU cycles, such projects are aimed at solving different class of problems. In these projects, users can act only as resource provider. User cannot run their own simulations. While previous all discussed systems and our work represents a set of systems that allow user simulations to run on remote machines.

CHAPTER III

S-DISPATCH ARCHITECTURE

S-Dispatch is designed to utilize any machine connected to the Internet. It allows participation irrespective of geographic location. It's architecture is based on the structured peer-to-peer substrate called Chord. This Chapter will discuss the design of S-Dispatch in detail.

A. Boot Strapping

It is similar to the node join protocol of Chord except for the number of joins made by each new node. It is assumed that the new peer knows the IP address of few existing peers in the framework by some external mechanisms. The new node will ask these existing peers to search for peers to populate its own neighbor list. Initializing neighbor list, updating existing node state and allocating keys along with associated data to new node are handled transparently at DHT level. Algorithms for all these operations are described in [22]. It takes $O(log^2 N)$ for *each* new node to join the system where N is the total nodes in the system.

Each node joins the identifier circle at several locations based on its computational capacity. The number of joins is equal to total number of physical CPUs times the computational capacity of each CPU. Each machine acts as several *virtual servers.* Thus if node joins identifier circle at M locations then total time for new node to join the framework will be $O(Mlog^2 N)$. This mechanism ensures that space occupied by each machine is proportional to its computational capacity.

B. Load Balancing

DHT's are normally used for applications where efficient data storage as well as retrieval is required. We use them in a unique way to distribute the task uniformly among all participating machines. Assume that the user i wants to run a simulation S and has a budget of B (number of CPUs required). Then the user will follow this steps:

1. It will generate B uniform random numbers $X_1, X_2, X_3, ...X_B$ between 0 and $2^m - 1$.
2. Each task is sent to successor of hash X_i
3. Each machine runs the simulation as an independent entity and report back to the task submitter at regular intervals of time the simulation result as well as current random number seed.

This mechanism ensures that the tasks are uniformly distributed among all available CPUs and load is proportional to the computational capacity of each CPU. If the remote machine or simulation crashes then it can be restarted on other machine with the last random number seed received from that machine. It is to be noted that our mechanism achieves load balancing in a *probabilistic* sense. Next section describes the process of forwarding the simulation result as well as random number seed to the task owner.

C. Result Aggregation

Each machine running the simulation is responsible for sending the simulation results at regular intervals of time. We refer to them as *update reports*. This allows access to intermediate simulation results to the task submitter. One way of doing it is to directly sent the result to the task submitter as done in SETI@home. But such an approach is not scalable as number of connections made to the task submitter is directly proportional to total number of machines running the simulations. For instance, if $50,000$ machines are running the simulations then task submitter will have to keep track of $50,000$ connections. Number of connections that can be opened at a time is limited by the OS. We can overcome this problem by increasing the granularity of sending updates i.e., increase the time interval between two updates. Such approach will solve the problem of opening too many connections at a time but does not guarantee that all the machines will send their updates. Machines used for running the simulation tasks are user workstations. So availability of such machines is restricted to few minutes to few hours. Thus if we increase the granularity to a large value then it is very likely that machine will not be available for so long or it crashes within that time period resulting into loss of simulation and no updates reaching the task owner. Thus we need a scalable, decentralized approach with fine granularity.

S-Dispatch takes advantage of routing algorithm of Chord. It takes $O(logN)$ steps, N being the number of nodes in the system, for searching the key in the network [22]. Remote machines running the simulation send their update report by searching the *node id* of task submitter. At each step, machine forwards the search query as well as the aggregated update report to its neighbor. Neighbor waits for all incoming reports and sends only one aggregated report to their neighbor along with search query. This process will continue until it reaches the task owner. Such an approach is scalable, decentralized and does not require large number of simultaneous open connections i.e., for N machines sending the updates, task submitter will have to manage $logN$ connections as compared to N.

CHAPTER IV

U-DISPATCH ARCHITECTURE

S-Dispatch represents a scalable framework where it can scale to millions of machines. It requires $O(log^2N)$ steps for joining network, $O(logN)$ steps for recovery when a node leaves the system and $O(logN)$ steps for collecting results. Such complexity and overhead is not desirable for environments where deployment is restricted to few hundred machines. For instance in research labs, business organizations etc. This motivated us to design U-Dispatch. It defines a set of protocols for sharing computational resources. Although it can scale to few hundred to thousand nodes only, protocols are decentralized and does not depend on geographic location of machine.

Peer who wants to join the network contacts *DisCache*, a well known peer whose IP address is provided with the software, for obtaining a group of participating peers in the system. Participant joins the framework by connecting this peer group. They act as *neighbors* for the new user. Search for idle resources and new resources in the network is done using *neighbor exchange* mechanism. Task allocation is done based on the *user budget* (number of CPU's needed for task) and the *current load* of each available machine. Preference is given to the machine with the least load. If the user budget is more than the available resources then all the resources will be used irrespective of the load. User will be allowed to add more resources as they become available in the network. Peer always try to remain connected to the system by updating its peer list at regular intervals of time and thereby improving the *resilience* of the network. This Chapter will describe all the above functionality in detail.

A. Node Join Protocol

This section will describe the process of joining and leaving the network. Participants can join the network to provide their idle CPU cycles to the other peers or can run their task on available machines. Both the activities are done voluntarily. Peers acting as resource provider can withdraw their participation from the framework whenever they want to.

Since U-Dispatch targets an open environment where any machine connected to the Internet can participate, process of joining the network should be decentralized, scalable and effective. Centralized indexes as used in napster [18] cannot be used in this case. U-Dispatch uses a unstructured peer-to-peer approach for joining new nodes to the framework. Unstructured peer-to-peer systems have desirable properties of being decentralized, scalable and affordable node join and leave cost. In most of the systems, node join is simple process of making few connections with the existing peers. It does not follow any strict rules nor does it try to maintain any relationship between topology and nodes as in structured peer-to-peer networks.

To join the network, user will follow the following steps:

1. User will contact *DisCache*, a well known peer, supplied with the client side software. DisCache caches the IP address of most recent peers that contacted it. It replies the query request by sending this list of possibly alive peers (since they joined the system last).

2. User will contact each of these peers and their current neighbors to built its own *neighbor list*

3. Before adding a new neighbor in its neighbor list, it will contact the remote peer to know its current status. Peer is added only if the remote machine is

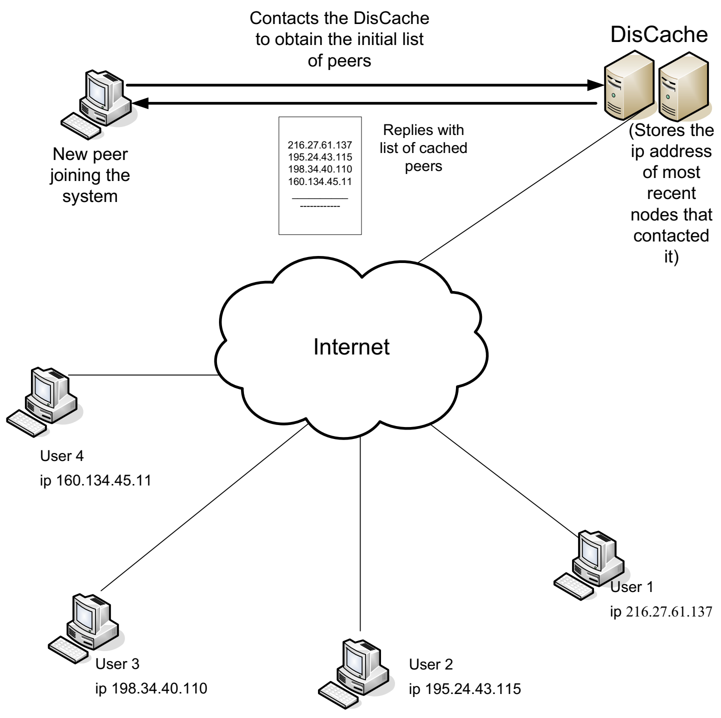
Contacts the DisCache to obtain the initial list of peers
DisCache
Replies with list of cached peers
216.27.61.137
195.24.43.115
198.34.40.110
160.134.45.11
New peer joining the system
(Stores the ip address of most recent nodes that contacted it)
Internet
User 4
ip 160.134.45.11
User 1
ip 216.27.61.137
User 3
ip 198.34.40.110
User 2
ip 195.24.43.115

Fig. 7. Node join protocol

available for use.

4. Steps 2 and 3 will continue until all the known peers are queried or when the participant has known sufficient number of available peers.

Figure 7 depicts these steps more clearly. User 1 joins the system first. It will obtain a list of peers that have join the system before from DisCache. At the same time, DisCache will store the IP address of user 1 i.e., 216.27.61.137. Now when user 2 joins the system and queries it for initial list of peers, it will receive all the IP addresses received by user 1 along with user 1 IP address. Same process will repeat for user 3 and user 4. DisCache will store only a *fix* number of IP addresses. Once that limit is reached, it will discard the *oldest* entries from the cache. Our node join protocol is simple, efficient, scalable and decentralized. It ensures that the *average* *degree* (number of neighbors) per node is not more than a small constant value. For e.g., Assume that DisCache stores x IP addresses. Now peer $A's$ IP address will be provided to first x peers joining the system after it. Afterwards, it's IP address will be discarded by DisCache as by that time it will have x new peer addresses cached in.

Once the peer joins the system, it will run a *maintenance* procedure at regular intervals of time in order to remain connected to the network. It will *ping* all its existing neighbors for their current status. If any peer has left the network, it will ask other neighbors for new nodes in the network. This is achieved by exchanging neighbor lists. At all time, each peer tries to maintain a group of available nodes in the system and thereby remain connected to the system. This makes the system very *resilient* and probability of node isolation very small.

B. Task Allocation

Task allocation is done based on the *budget*, number of CPU requested for task, and the *current load or weight* of each available peer. Load is defined as the ratio of computational capacity of the machine and total number of tasks running on that machine. We aim to develop a task allocation scheme that is able to distribute the task in optimal fashion so as to prevent overloading of any single machine and at the same time minimize the computational time. We do not want any *centralized scheduler* for task scheduling and so the users can run their task immediately in real time. We ensure that users do not need to reserve resources for future use nor they need to submit their task to some centralized system as in Condor and XtremWeb. Before we describe our task allocation strategy, it important to understand the interpretation of *available* machines for our system and how they are detected in our framework.

1. Finding Idle Machines

Available machines are defined as the machines which provide their *idle* CPU cycles for running remote task. Different systems use different strategies to detect these idle machines. For instance, in XtremWeb idle machines declare their own availability to the root servers. In OurGrid and Condor, centralized servers are responsible for detecting idle machines (see related section for more details). We use the *login* program to detect idle machines in the network. Any machine that is either in the *log off* state or with a *locked screen* will be detected as idle machine. Idle machines will be allocated irrespective of the user settings. We differentiate between *ownership* and *allocation* of the computational power. A machine is donated by its owner without giving up the ownership. Although the donated capacity of the machine is temporarily allocated to users who do not own it, each individual resource remains under the control of its owner. The owner can regain the allocation of the resource whenever needed. When the user logs on to the machine, it will be automatically removed from the system. By this mechanism, owner is insured that the machine would be utilized only when no interactive work is done on it. We need to allocate the available resources in an optimum manner so as to maximize the system utilization to reduce the computational time.

2. Task Allocation Algorithm

While allocating task, preference will be given to the least loaded machine i.e., machine with the highest ratio. Our architecture follows the following steps for allocating the task.

1. Flood the network with a certain TTL (Time to Leave)

2. Get the weight and total number of *physical CPUs* from each responding host

(If machine is not available set weight as ∞)

3. Get the budget from the task submitter.

4. Sort all the available host in decreasing order of weights and send them the simulation in that order until user budget is exhausted.

5. Number of simulation copies offloaded to each machine will be equal to number of physical CPUs. Each copy will act as an independent simulation.

This algorithm first loads the fastest machine, then the slower one. It also prevents new hosts from using overloaded hosts and allocates tasks so as to minimize the total computation time. Architecture allows user to include resources dynamically as they become available. User do not need to stop the task and re run it to utilize the newly available resources. Offloading number of tasks equal to total number of physical CPUs ensures that all the available capacity is utilized efficiently. Due to the flood-based design, U-Dispatch does not scale well beyond several hundred or thousand users However, it allows a more optimal (i.e., deterministic) task allocation based on the current load and CPU capacity than S-Dispatch, which achieves optimality only in the probabilistic sense. Also it has less overhead than S-Dispatch.

C. Result Aggregation and User Control

U-Dispatch provides real time execution environment. Like Condor or XtremWeb, user do not have to submit their job and then wait for centralized servers to schedule them. User in U-Dispatch will be able to run their tasks immediately and interact with them at run time. Remotely running simulations will send the intermediate results in an *update report* at regular intervals of time to the user who started it. U-Dispatch provides facility of processing this intermediate results received from all the remote machines and displaying the final result to the user. User can stop the simulations once the desired results are obtained. User can also pause and resume their simulation on remote machines whenever they desire. Similarly, the owner of the resource can pause, resume and stop all the remote simulations running on the resource whenever needed. Next chapter will describe this functionality of U-Dispatch in more detail.

CHAPTER V

U-DISPATCH IMPLEMENTATION

The previous Chapter discussed about U-Dispatch from the architectural point of view. This Chapter explains the framework from user point of view. We first talk about the various components of U-Dispatch along with their functionalities. We then discuss the process of writing a simulation task in U-Dispatch and explain it with a sample simulation and finally we discuss the security features of U-Dispatch.

A. U-Dispatch Components

Current implementation of U-Dispatch is restricted to Window OS. It can be deployed on all versions after Windows 2000. From the user's perspective, U-Dispatch consist of three major components 1) A U-Dispatch Service 2) U-Dispatch GUI and 3) U-Dispatch Controller. All these components are explained below in detail. If U-Dispatch is used only as resource provider then no additional softwares are needed but if it is used for consuming remote resources then it will require *microsoft visual studio 2005* or *microsoft visual studio .net 2005* installed locally.

1. U-Dispatch Service

U-Dispatch Service is a multi threaded windows service written with Win32 API. It is the main component of the framework. User can participate in U-Dispatch framework only if service is running on the local machine. From the resource provider point of view, it is responsible for authenticating the remote user and starting their task on the system, keeps track of all the tasks started on the system along with their control information like task comments (if added), user name, time and date information and the current state of each task (paused or active). U-Dispatch provides a real time

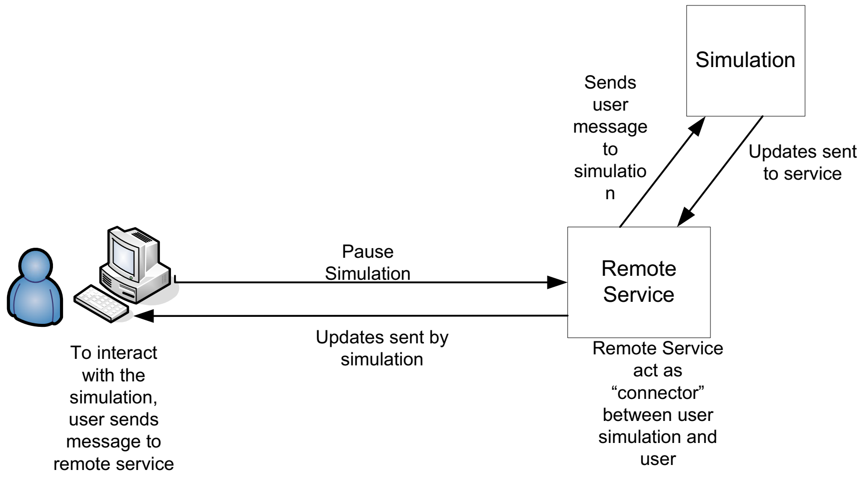
Simulation
Sends user message to simulatio n
Updates sent to service
Pause Simulation
Remote Service
Updates sent by simulation
To interact with the simulation, user sends message to remote service
Remote Service act as "connector" between user simulation and user

Fig. 8. User interaction with remote simulation task in U-Dispatch

interactive environment for user running the simulation task. User interactions with the remotely running simulation task takes place through the service running on that machine. Thus, service becomes a single point of contact between the participant and the remote task. For e.g., As seen in Figure 8, if a user wants to pause the simulation then it will send the "pause" message to the remote service. On receiving the message, remote service makes sure that message is sent to the appropriate simulation task. Also all the messages sent by the simulation are received by local service and it is responsible for delivering the message to the appropriate task submitter.

From the resource consumer point of view, service is responsible for keeping the machine connected to the framework. Initially it will contact *DisCache* to obtain the list of peers so the machine can join the network. Afterwards it will periodically update its neighbor list to purge out dead machines (machines that left the network) and add new available machines. It also helps the user in finding and adding new resources dynamically to start more copies of the simulation.

2. U-Dispatch GUI

U-Dispatch GUI application represents the user interface for the participants. It is written in Microsoft Foundation Class (MFC) and is compatible with visual studio 2005 and visual studio .net 2005. Unlike other projects, where simulation tasks are independent and written separately and then submitted to the framework, in U-Dispatch user writes simulation into U-Dispatch GUI application (explained in next section). GUI application acts as *frontend* for users to interact with their simulations and it also acts as *simulation process* for remote machines. Once simulation is written into the application, user only have to run the application. Before running the application, user needs to make sure that U-Dispatch service is started. From the GUI, user will be allowed to set various parameters for running the simulations. These parameters include whether the simulation needs to run remotely or locally, setting the budget (number of CPU's needed) for the simulation and comments about the simulation. If remote mode is selected then simulations will be automatically distributed on remote machines based on the budget given by the user. If budget is more than the available resources then all the resources will be allocated. User will be allowed to add more machines, as they become available, to run additional copies of task. As discussed before, preference will be given to the machine with the highest "weight" ratio i.e., least loaded machine. GUI application is also responsible for collecting the update reports sent by remotely running simulations, for processing them as requested by the user and displaying the final result to the user. Next section will talk about update reports in more details.

Once the simulations are started remotely, user will also be able to interact with them by pausing, resuming the simulation from paused state and stopping them. All such events, either performed locally or remotely, will be reported back to the GUI application. Events will also include simulation crashes. If any task fails or crashes due to some reason then it will be reported back to the application. Users will also be able to interact with each remote node (where the simulation is running) to get the information about the competing simulations (if any), who started it, comments on the simulation (if included), number of physical CPUs along with their computational speeds. User will also have access to *local view* of the system. It includes all the machines where the user simulation is running along with the neighbors of each those nodes. User can get an idea of *topological structure* of the network with the help of the local view.

3. U-Dispatch Controller

Our idle CPU detection algorithm takes into consideration all the machines that are either in the “log off” state or machines with “locked” screen. Now it is possible that the machine is hosting some remote task when a user logs back into the machine. As a result, user will not be able to utilize all the CPU cycles of the machine. To overcome this problem, we designed and implemented a stand alone application called the *U-Dispatch Controller*. U-Dispatch Controller keeps track of all the task running on the local machine. User can pause, resume or stop any of the remote task as per the need. All such events will be notified to the task submitter. Here again, U-Dispatch service becomes a “bridge” between task submitter and U-Dispatch Controller. U-Dispatch Controller notifies the local service about the action to be taken on certain simulation. Service is responsible for taking that action on simulation and for notifying the appropriate user about the action and for withdrawing the participation from the framework (since user has logged back into the local machine).

B. Writing Simulations

Writing simulations is very different from any of the existing projects. To better understand the process, we will explain it with one simple problem.

1. Birthday Problem

Suppose we have a group of 30 people in a class room and we want to find out the probability of two people sharing the same birthday. This is a simple problem and can be solved directly using probability theory. But here we will try to solve this problem in distributed fashion. We have deliberately taken such simple example so that process of writing simulations in U-Dispatch can be demonstrated effectively.

2. Solution Approach

Each machine running the simulation will perform the following steps:

1. Pick 30 uniform random numbers in the range [1,365]. Each number represents one day of the year.
2. Check to see if any of the thirty numbers are equal. If equal then increment the match value
3. Increment the iteration value
4. Go back to step 1 and keep repeating it until the timer expires.
5. Report the total match value and total iterations performed till that time. This is called sending *update reports*(see Figure 9).
6. Go back to step 1.

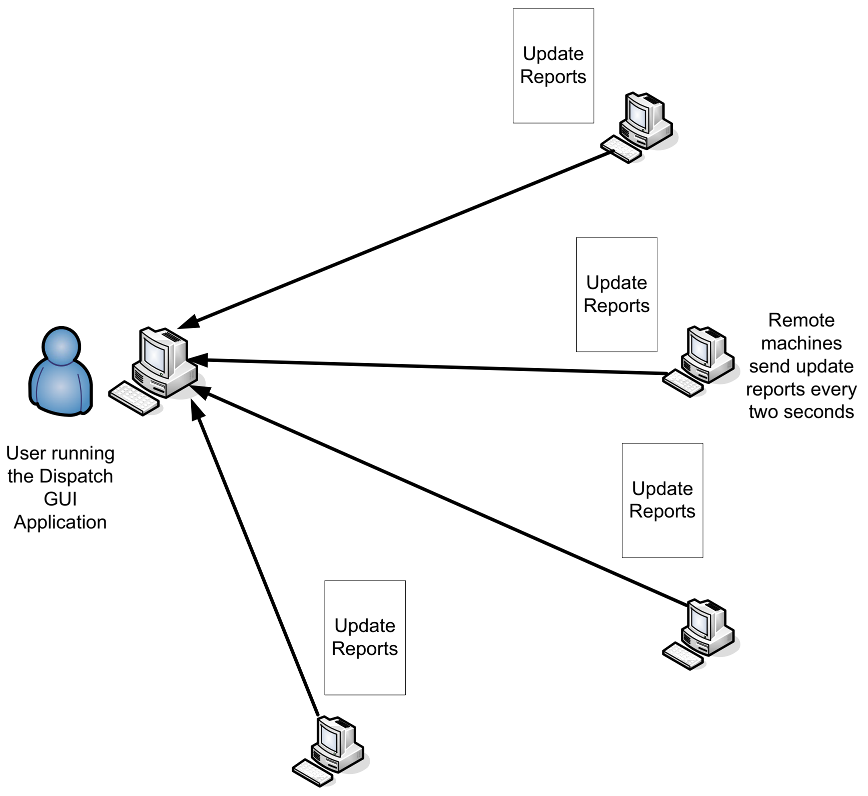
Update Reports
Update Reports
Remote machines send update reports every two seconds
User running the Dispatch GUI Application
Update Reports
Update Reports

Fig. 9. Remote machines send update reports every two second. Reports contain the total match value and total iteration value performed till that time

The above solution will run for infinite number of iterations. Typically user will stop the simulation process after the desired results are obtained. GUI application will receive the update reports from all the machine as shown in Figure 9. It will add all the match values and all the iteration values and report the fraction of iterations that have matching birthday back to the user.

3. Solution in U-Dispatch Framework

To write any simulation in U-Dispatch, user needs to modify three files in the GUI application, simulation.h, simulation.cpp and report.h. Simulation needs to be written in C++ format. All the variables and the member functions pertaining to simulation

```
//Report.h File
class report
{
public:
      //Variables required for processing results are defined in
        report.h
      __int64 match;
      __int64 itr;

      report();
      void send_report( void* r, int size);
};

//simulation.cpp file. Algorithms are written in procedure function
void simulation::procedure(LPVOID obj)
{
      report *rep;
      rep = new report();    //generates new report
      int t1 = GetTickCount();  //initialize the start time
      init_genrand ((unsigned long)time(NULL)); //initialize random
                                                  number generator
      while(true)
      {
      //Generate 30 random numbers and insert it into set STL
      //days. If the size of days is less than the value of k, it means
      //number is repeated and match is found
            for(int k = 1; k<31; k++)
            {
                  double d = genrand_real1() * 364 + 1;
                  days.insert(d);

                  if(days.size() < k)
                  {
                        rep->match++ ; //increment the match value if
                                       //day is repeated.
                        break;
                  }
            }

            rep->itr++; //Increment the iteration value
            days.clear(); //empty the set for next round.
            DWORD t2 = GetTickCount();//Get current time.
            DWORD elapsed = t2 - t1;

            if(elapsed >= 2000) //checks the timer. If timer expires,
                                //it sends report to the host
            {
                 t1 = t2;
             //Report is sent using send_report(void*,int size) api.
                  rep->send_report (rep, sizeof(report));
       //Cheks if simulation is paused remotely or locally
      if (WaitForSingleObject(pmain->handle, INFINITE)!= WAIT_OBJECT_0)
                        break;
            }
      }

}
```

Fig. 10. Report.h and simulation.cpp files for birthday problem

should be defined in simulation.h. All the member function body should be included in simulation.cpp. simulation.cpp also contains two additional member functions, procedure and interpret. The code that executes the algorithm should be included into procedure function and code that interprets the results obtained from the remote nodes should be included in interpret function. Variables that are needed back for processing in interpret function are included in report.h.

As seen in Figure 10, *match* and *itr* values that are needed back for processing are

```
//update report interpretation logic is written in interpret function
void simulation::interpret()
{
      //Get the total number of remote nodes
      int total_nodes = pmain->get_total_nodes();


      //Local variables needed for processing output.
      __int64 total_match = 0;
      __int64 total_itr = 1;


      report *r;

      CSingleLock s(&(pmain->sema));
      s.Lock();
      for(int i=0; i < total_nodes; i++)
      {
        //Get the update report of node with node id i
            r = (report *) pmain->get_update (i);

            total_match += r->match;  //Add all the match values
            total_itr += r->itr;     //Add all the iteration values

      }
      s.Unlock();

      double ans = (total_match * 1.0)/(total_itr); //Generate the
                                                     //  result

      //Results are printed onto the GUI using print function.
      pmain->print("Probability = %.15f Total itr = %I64d \n",
ans,total_itr);

}
```

Fig. 11. Interpret function for birthday problem

included into report.h file. Code that executes the algorithm is included in procedure function. As a first step a new report object is instantiated. Thirty uniform random numbers ([1,365]) are generated and are inserted into Standard Template Library (STL) set days. Set has a property of storing only unique values. If a value is repeated then it is ignored by the set. We use this property of set for counting the number of matches found. After each random number is inserted in the set, we check the size of the set. If the size of the set is less than the total number of random numbers generated, we increment the value of match. Process is repeated until the timer expires. Once the timer expires, total match count and total iterations performed till that time are sent to the local service using the send_report(void*,int size) API. Local service will U-Dispatch the update results back to the GUI application of the job submitter. All the remote machines will perform this operation until user does not pause or stop the simulation.

Once the GUI application receives update from all the remote machines, it will process the update based on the *processing logic* provided in interpret function. As seen in Figure 11, we first get the total number of machines where simulations are running using get_total_nodes(void). Then, in the *for* loop we get the latest update report of all the machines using get_update(int nodeId) function. Finally, the results are printed on the GUI application using print function. The arguments to print function are similar to the C style printf function. pmain is the handle to main application.

C. Security

U-Dispatch provides anonymous access to the remote machines. Remote users are not aware about the identity of the participant whose task is executing on their machine. Such system requires strict security policies so as to prevent compromised user from running the task and from preventing remote jobs to access the personal information of remote machines. This section talks about the security aspect of U-Dispatch.

In a peer-to-peer environment, there are no centralized servers with security databases that can provide typical security services such as authentication and authorization. For example, in an Active Directory domain, domain controllers provide authentication services using Kerberos. In a server less peer environment, the peers must provide their own authentication. This essentially means that peer machines have to be configured properly so as to provide safe and secure environment.

Windows operating system provides a rich set of API to configure local system security. We exploit these APIs to configure the local machines. To better understand the topic, we will concentrate on two peer system, *client* and *server* peer. Client peer acts as resource consumer and server peer acts as resource provider. We assume that

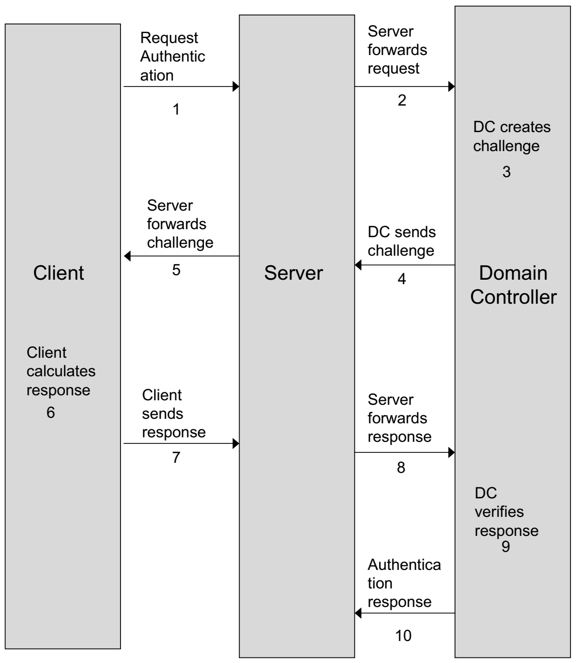

Fig. 12. NTLM authentication process

resource is idle and is available for use and client requests for it. Defining client and server peer is only a modeling assumption and has no relation with traditional client/server model discussed before.

1. Authentication in Single Domain Network

If U-Dispatch is deployed on machines belonging to same domain then authentication is provided using NT LAN Manager (NTLM) protocol. NTLM is the existing security protocol, for remote client authentication, of Windows.

NTLM protocol is fairly straightforward and is based on what is known as a challenge/response sequence. The sequence illustrated in Figure 12 works like this:

1. The client sends its username and domain name to the server.

2. The server forwards this information to the domain controller (DC).

3. The DC creates a *challenge*, which is randomly generate and created with the client's password (known only by the client and the DC).
4. The DC sends the challenge to the server.
5. The server forwards the challenge to the client.
6. The client examines the challenge using its password and performs a *known modification* to the challenge, creating a *response*. The known modification is defined by NTLM protocol.
7. The client sends the response back to the server.
8. The server forwards the response back to DC.
9. The DC examines the response and verifies that it is the original challenge modified using the known modification. The client is now authenticated.
10. The DC informs the server that the client is authenticated.

All the steps discussed above are done implicitly using several Win32 API's. Client or server do not need to provide any information explicitly. If authentication fails then client will not be able to use the remote CPU.

2. Authentication in Multi Domain Network

If U-Dispatch is deployed on machines belonging to different domains then NTLM protocol will not be able to authenticate a client belonging to different domain than that of a server. A different mechanism needs to be deployed. Different systems implement different mechanism for achieving this. For instance, Condor uses a policy based access mechanism. Each user provides the system list of allowable domains.

When a client connects to the server, server checks its access policy to see whether client is allowed to use the resource or not. In Xtrem Web and OurGrid, centralized servers are responsible for access check. Other examples include use of digital certificates, public key algorithms like RSA etc.

In U-Dispatch, we do not use any of the above mechanism but instead we configure each machine appropriately so as to provide secure execution. Before we get into details of how U-Dispatch configures local machine, we need to understand the meaning of *Tokens* and *Impersonation* that are explained next.

a. Tokens

User normally provides a username and a password to interactively log on to a workstation or server machine running Microsoft Windows Operating System. The username is the trustee account that exist either in Active Directory located on a domain controller or as a local user account maintained in a security database on the local machine. The password is the user account's credentials, which the system uses to *authenticate* the logon session.

After the user logs on, system creates a list of privileges that are assigned to the user account as well as all the user's group accounts. The system also gathers other information about the user account like the local group accounts of which the user is a member, domain-level information such as the domain groups of which the user is a member, credentials, security descriptor etc. All this information is compiled into a data structure that is maintained in a system kernel object, which is called the *token.* After the system has authenticated a logon attempt, token *is* the user's identity as far as Windows security is concerned.

b. Impersonation

Impersonation is the ability of a thread to execute in a security context that is different from the context of the process that owns the thread. It is extensively used in traditional client/server architecture. Once the client is connected to the server, threads in server process run under the client's security context or credentials and so the process will only have access rights of the client even though it is started by server. When the thread no longer needs to behave as though it were another user, it can revert to its normal state i.e., they can start running under the security context of server. Impersonation allows restricted access to the resources of the server machines.

U-Dispatch takes advantage of impersonation and user account token to configure local machine. It follows the following steps to ensure the secure execution of remote task of an anonymous user:

1. When ever peer wants to participate in the framework, a new account called U-Dispatch is created in the machine.
2. All the access rights i.e., read, write and delete are removed from that account.
3. U-Dispatch Service receives the simulation in a separate thread.
4. Thread will create a token for U-Dispatch account.
5. Using that token, it will impersonate itself as U-Dispatch account and run the simulation under the security context of U-Dispatch account.

This mechanism ensures that simulation executable running on remote machines do not have any access to the data stored locally. It cannot modify or delete any information nor it can read any personal information stored on the machine. One important point to note here is that the simulation will be able to create files on local drives but will not be able to write into those files. This is due to the fact that access credentials of local drive allows all the local users to create files on it. Since U-Dispatch is nothing more than a dummy account, it will also get that access right by default. But the access credentials of U-Dispatch account will prevent it from writing into any files even if it is created by the account itself. Thus, simulation will only be able to access the computational power of the system and nothing else.

D. Working of GUI Application

In this section, we explain the implementation details of U-Dispatch GUI application. It is a multi threaded application written in Microsoft Foundation Class (MFC). One of its unique feature is that the same application acts a *GUI* for task submitter and as a *remote simulation* for remote service. The differentiating point is the information provided with the application as the *command line argument*. When the user or remote service runs the application, it will first check the command line arguments parameters passed to it. If no arguments are passed to it then it acts as GUI application otherwise it acts as simulation process. Based on this decision, different actions are performed by the application. For instance, if executable acts at GUI application, thread invoking procedure will not be forked. Similarly if executable acts as simulation then thread invoking interpret will not be forked.

When the application acts as U-Dispatch GUI, the following steps take place:

1. Application connects local U-Dispatch Service and obtains the list of remote peers.

2. It contacts each remote machine to obtain their current machine state(busy or free), number of physical CPUs, weight ratio. It sorts all the *free* machines based on decreasing weight ratio.

3. GUI application executables are dispatched to remote machines based on the obtained information and user budget along with the control information. Control information includes the host IP and port address for receiving update reports, simulation comments if included and username.

4. Simultaneously two threads are forked. First thread called the *service thread* is responsible for receiving update reports from all remote machine and for notifying events that take place on simulation either locally or remotely. Second thread will invoke interpret function at periodic intervals of time (typically two seconds). As explained before, interpret function will process the results and show them to the user.

GUI application will act as simulation binaries for remote machines. On receiving such files, remote service will take the following action:

1. Service authenticates the task submitter. This step takes place only if both remote user and task submitter belong to same domain as explained before.

2. It will store all the control information sent by the user regarding the simulation.

3. If both the steps occur correctly then service will receive the actual executable file.

4. Before running the executable file, service will *impersonate* the security context of the thread to that of U-Dispatch account and pass a *random port address* to the executable as an *argument*. Simulation task will listen on this port for the event notifications from the service.

5. Service will run the simulation under the security context of U-Dispatch account.

6. Since it is running as a simulation (as application will receive port address as argument), it will fork another thread that will call the procedure function.

7. Procedure function will execute the algorithm and is responsible for sending update reports to the remote GUI application through local U-Dispatch service as explained before.

This type of implementation where same application acts as front end to the user and also acts as simulation is quite different from other projects. Most projects discussed earlier have simulation task as a separate entity. Our design have many advantages over them. Firstly it helps in providing real time execution environment where user has access to intermediate results and simulation running on remote machines. Unlike batch system where user has to wait for the task to get over, user in U-Dispatch can stop simulation at any time once desired accuracy is obtained. Secondly, other projects require remote users to share local storage devices to store the simulation result. User may not be comfortable in sharing the local drives to remote user without knowing their identity. Therefore such systems require strict security policies, strict authentication and other various mechanisms to prevent malicious user to run their task. Our design does not require user to share local storage devices. It only uses remote machine's computational power. All the intermediate results are stored in memory buffers and are dispatched to user at regular interval of times. Thirdly, it makes the process of collecting results and interpreting the results far more simple and dynamic. In systems discussed in related work, user has to collect all simulation results from remote machines manually and then write another application to process them. U-Dispatch combines both this activities into one. Codes for executing algorithm as well as for interpreting the results are written in the same file and are called accordingly.

CHAPTER VI

EXPERIMENTS

In this Chapter we describe the experiments performed in the Internet to check the performance of U-Dispatch. First, we try to understand the performance gain achieved by running the simulation in parallel on large number of machines having heterogenous computational capacity. Secondly, we check the performance of U-Dispatch on a network consisting of machines having same computational capacity. This gives us an insight on the *actual* performance gained achieved by increasing "x" machines. Next we deploy U-Dispatch on a multi domain network where machines are geographically apart and belong to different domains. This Chapter describes all the experiments in detail along with the results.

A. Test Simulation

For all the experiments, monte carlo method of computing the value of π is taken as a test simulation. Value of π can be calculated by throwing random points on a 1×1 square and counting how many of them falls into the circle of radius 1. To accomplish this, generate two uniform random numbers x, y $(0 \leq x, y \leq 1)$. If $\sqrt{x^2 + y^2} \leq 1$, the point is considered to be inside the circle (otherwise, outside the circle). It is clear that the probability p that a random point falls inside the circle is $\pi r^2/4 = \pi/4$ since $r = 1$. By computing p in simulations, we can deduce an approximation to π as $4p$.

To calculate the value of p in distributed fashion, we distribute the simulation to remote machines. These remote machines run the simulation and send the *update reports* after every two seconds. Update reports consist of the total number of iterations and the total success count (iterations in which points falls within a circle) performed until that time. All the reports are aggregated at a single machine. Let

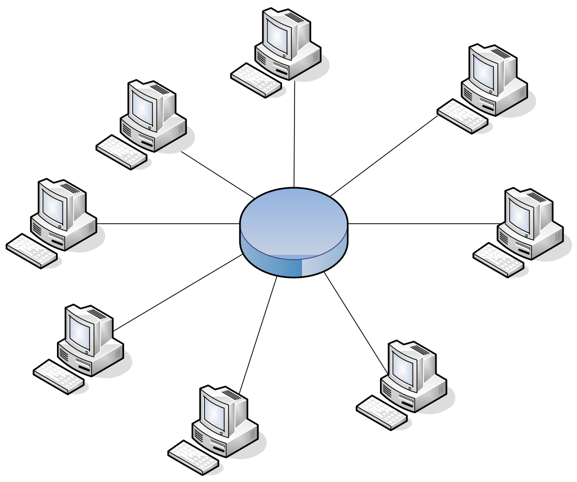

Fig. 13. Physical topology for experiment. All 130 machines are connected to same switch

$success_t$ be the sum of all the success value received so far and let itr_t be the sum of all the iteration values received so far from all the machines. Then the value of p is calculated as $p = sucess_t/itr_t$ and value of π is calculated as $4p$. See Appendix A for code in U-Dispatch

B. Experiment Setup

Experiments are carried out in emulab [8]. All the machines are attached to a single switch with 100 Mbps links in a ring topology as shown in Figure 13. 130 machines are used in the experiments. All the machines are workstation with single physical CPU having heterogenous computational capacities. Simulation is started by one of the machines and distributed to all the machines. Current implementation of U-Dispatch allows using immediate neighbors only. For experiment purpose, each node in the framework knows about all the other nodes in the system. This allows us to run the experiments on large number of machines simultaneously. To achieve this, we implemented a different node join method which is explained next.

1. Modified Node Join Protocol

We want each machine to know about all the other peers in the framework and we want to achieve that in a simple scalable manner. We take advantage of *IP multicast* technology.

Definition 2. *IP multicast: A IP multicast is a single stream of data (i.e., a set of packets) that is transmitted simultaneously to selected multiple hosts who have joined the appropriate multicast group. In contrast to broadcasts, multicast clients receive the data stream only if they have previously elected to do so (i.e., by joining the specific multicast group address).*

Any peer that wants to join the framework joins a pre-defined multicast group and declares its availability by advertising its IP address to the group. Thus all the existing peers will know the IP address of new peer joining the framework. It is to be noted that such method of peer discovery will work because all the machines belong to the same domain. It will not work if the machines belong to different domains since routers in the Internet are disabled by default for multicast traffic. Routers will drop the packets having multicast address as the destination address. When a machine in emulab boots up, it will automatically start U-Dispatch Service. Service will join the multicast group and declare its availability to other existing peers. Same procedure is repeated by all the machines in the framework. Except for node join protocol, all other functionality of U-Dispatch remain unchanged.

C. Experiment Results

We aggregate all the results at a single machine and calculate the total iterations performed by adding all the iteration values received by that time. We plot these values against number of machines used in the experiment. From Figure 14, we can

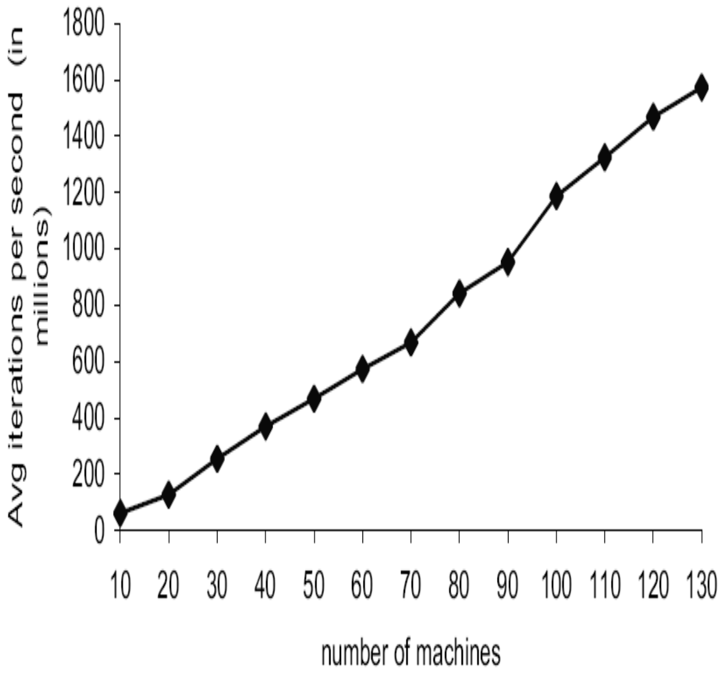

Fig. 14. Average iterations in millions per second.

see the performance improvement achieved by increasing the number of machines. For each entry on x-axis, number of iterations/sec are averaged over a forty five minute run of the simulation. For 130 machines, close to 1572.678 million iterations are performed per second.

In the previous experiment, it was difficult to compare the *actual* performance improvement achieved by increasing x number of machines. This was attributed to the fact that machines with different configurations were used and simulations were distributed to arbitrary machines in an arbitrary fashion. So, to understand the actual performance gain, simulation needs to run on the machines with similar configuration.

Figure 15 shows the results of such experiment. This experiment was carried out on eighty machines. All the machines had the same configuration. Again for each entry on x-axis, the experiment was carried out for forty five minutes and average

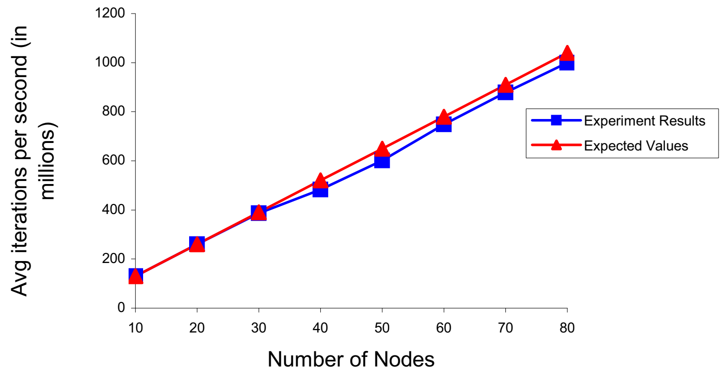

Fig. 15. Average iterations in millions per second. Simulations performed on machines having similar configuration

number of iterations/second are plotted. On one machine, it performed on an average thirteen million iterations per second. Based on that value, expected values are plotted. Network is scaled from ten machines to eighty machines. From the plot, we can see that experimental values and expected values overlap each other until thirty machines. After that as the network is scaled, the gap between expected value and experimental values starts to widen up. This is due to the fact that all the machines start simulation at different times and send their updates at different times. So whenever the code that interprets the results is evoked (after every two seconds), all the machines might not have send their *latest* updates by that time. Up to thirty machines, there is no effect of this timing difference but after that as the network is scaled, it becomes prominent. For eighty machines, it performed close to 994 million iterations per second.

D. Wide Area Network Experiment

All the above experiments are performed on machines belonging to same domain. We also successfully check the performance of U-Dispatch when it is deployed on machines belonging to different domains and are geographically apart. Experiment is carried out in Internet Research Lab (IRL) at Texas A & M University. Eight machines, having heterogenous computational capacity and different number of physical CPUs, are used in the experiment. Lab members start the experiment from their home machines. One of the machine present in the lab act as *DisCache* while the other machines act as resource provider. User contacts DisCache and obtains list of all available resources. User was able to start the simulation on all the available machines successfully. Also, user was successfully able to starts a experiment from a lab machine on all the other machines in lab and on the home user machine that was participating as resource provider.

CHAPTER VII

CONCLUSION AND FUTURE WORK

A. Conclusion

In this Chapter, we summarize our work and the results we obtained. Also we discuss the future work for improving design of Dispatch

We designed two different architecture based on target deployment and implemented one of them for sharing computational resources to run the simulation task in parallel. S-Dispatch can scale to millions of node but is complex and has a large overhead for its several protocols. As compared to that U-Dispatch is much simpler to deploy but can only scale to few hundreds of machine. In both framework, node join protocol is decentralized and scalable. It allows users to participate in the network irrespective of their geographic location. Task allocation scheme ensures optimal selection of available resources to run simulation task. Also both frameworks provide user the flexibility of deciding budget for their task. It also allows dynamic addition of more resources as they become available. It allows user to interact with each simulation task in real time. Task submitter can pause, resume or stop any instance of simulation when ever needed. Our security model ensures that remote tasks are not able to access any resource of the machine except computational capacity.

Writing simulations in U-Dispatch GUI is made as much flexible as possible. Several user friendly API's are available to users that simplifies this process. U-Dispatch Service ensures that machine remains connected to the framework at all times. It is also responsible for authenticating and starting remote tasks. U-Dispatch Controller is responsible for providing localized control to the user whose machine is used as resource provider. It allows user to stop or pause and resume facility for all the tasks running on local system.

U-Dispatch was deployed and tested on a large network consisting of $10 - 130$ machines. Results show the performance gain achieved by using Dispatch for running tasks in parallel. We also successfully tested U-Dispatch in a wide-area network where it is deployed on machines belonging to different domains.

B. Future Work

We would like to implement S-Dispatch and check its performance when deployed on large number of machines and compare it with U-Dispatch. Currently U-Dispatch is used in IRL (Internet Research Lab). We would like to go one step further and deploy it in CS labs and check the performance when deployed on few hundred machines.

REFERENCES

[1] BitTorrent. [Online]. Available: http://www.bittorrent.com, August 12, 2007.

[2] Berkeley BOINC. [Online]. Available: http://boinc.berekeley.edu, August 12, 2007.

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

[4] W. Cirne, D. Paranhos, L. Costa, E. Santos-Neto, F. Brasileiro, J. Sauv, F. A. B. da Silva, C. O. Barros, and C. Silveira, "Running Bag-of-Tasks Applications on Computational Grids: The MyGrid Approach," in *Proc. ICCP'2003 - International Conference on Parallel Processing*, Oct. 2003, p. 407.

[5] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Wide-area cooperative storage with CFS," in *Proc. SOSP'01*, Oct. 2001, pp. 202–215.

[6] Distributed.net. [Online]. Available: http://www.distributed.net, August 12, 2007.

[7] P. Druschel and A. Rowstron, "PAST: A large scale, persistent peer-to-peer storage utility," in *Proc. SOSP'01*, Oct. 2001, pp. 329–350.

[8] Emulab. [Online]. Available: http://www.emulab.net/, August 12, 2007.

[9] G. Fedak, C. Germain, V. Néri, and F. Cappello, "XtremWeb: A generic global computing system," in *Proc. CC-Grid 2001 Special Session Global Computing on Personal Devices*, May 2001, pp. 582–587.

[10] A. Geist, A. Geguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *PVM: Parallel Virtual Machine.* MIT Press, Cambridge, 1995.

[11] Gnutella. [Online]. Available: http://www.gnutella.com/, August 12, 2007.

[12] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface.* MIT Press, Cambridge, 1999.

[13] 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, pp. 37–42.

[14] IBM. [Online]. Available: http://www.ibm.com/developerworks/grid/newto, August 12, 2007.

[15] KaZaA. [Online]. Available: http://www.kazaa.com/, August 12, 2007.

[16] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao, "OceanStore: An architecture for global-scale persistent storage," in *Proc. Proceedings of ACM ASPLOS*, Nov. 2000, pp. 190–201.

[17] M. Litzkow, M. Livny, and M. W. Mutka, "Condor - A hunter of idle workstations," in *Proc. 8th International Conference on Distributed Computing Systems (ICDCS 1988)*, June 1988, pp. 104–111.

[18] Napster. [Online]. Available: http://www.napster.com, August 12, 2007.

[19] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content-addressable network," in *Proc. ACM SIGCOMM*, Aug. 2001, pp. 161–172.

[20] A. Rowstron and P. Druschel, “Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems,” in *Proc. IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, Nov. 2001, pp. 329–350.

[21] SETI@home. [Online]. Available: http://setiathome.berkeley.edu/, August 12, 2007.

[22] 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*, Aug. 2001, pp. 149–160.

[23] R. Zhang and Y. C. H. 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),Monterey,CA*, 2003, pp. 172–179.

[24] B. Y. Zhao, J. D. Kubiatowicz, and A. Joseph, “Tapestry: An infrastructure for ault-tolerant wide-area location and Routing,” Univ. of California, Berkeley, Tech. Rep., 2001.

APPENDIX A

SIMULATION CODE IN DISPATCH FOR EXPERIMENTS

```
//report.h
class report
{
public:
    __int64 success; //variables required for processing
    __int64 itr;
    report();
    void send_report( void* r, int size);
};
//simulation.h
class CDispatchDlg; //forward declaration for CDispatchDlg class
class simulation
{
public:
    //member functions required by all simulations
    CDispatchDlg *pmain;  //handle to the main application
    simulation(LPVOID mainwnd); //constructor for simulation class
    void init ();
    void procedure(LPVOID obj); //member function where code executing
                                //algorithm is written
    void interpret(); //member function for writing processing logic
    //Variables required for this simulation
    bool first_interpret;
    DWORD sim_started;
    __int64 prev_total_itr;
};
//simulation.cpp file
#include "stdafx.h"
#include "Simulation.h"
#include "RandomNumGenerator.h"
#include "Report.h"
#include "DispatchDlg.h"
simulation::simulation(LPVOID mainwnd)
{
    pmain = (CDispatchDlg*)mainwnd; //initialize the main window handle
    prev_total_itr = 0;
    first_interpret = true;
}
// runs only on a local computer; disabled on a remote system
void simulation::init()
{
    //AutoRun Functionality
    //Select Mode: 0 for locally
    //        1 for remotely
    //    2 for both
    //pmain->SetSimMode(2);
    //pmain->PostMessage(WM_COMMAND, IDC_RUN, BN_CLICKED);
```

```
}
void simulation::procedure(LPVOID obj)
{
    report *rep;
    __int64 last_itr = 0;
    rep = new report(); //create a new report object
    int t1 = GetTickCount(); //initialize timer
    init_genrand ((unsigned long)time(NULL)); //initialize the random number
                                               //generator
    while(true)
    {
        double x = genrand_res53(); //Generate random number
        if (x <= 1)
        {
            double y = genrand_res53();
            //Check whether the random numbers are within circle or not
            if(x*x + y*y <= 1)
            rep->success = rep->success + 1;
        }
        rep->itr++; //Increment the iteration value
        // do not check the clock too often
        if (rep->itr - last_itr > 1000)
        {
            DWORD t2 = GetTickCount();
            DWORD elapsed = t2 - t1; //check the timer
            if(elapsed >= 2000)
            {
                //send the report if timer expires
                t1 = t2;
                rep->send_report (rep, sizeof(report));
                //check if any event took place on
                //simulation (paused or resumed)
                if (WaitForSingleObject(pmain->handle, INFINITE)
                    != WAIT_OBJECT_0)
                    break;
            }
            last_itr = rep->itr;
        }
    }
}
void simulation::interpret()
{
    int total_nodes = pmain->get_total_nodes();//Get total remote
                                                 //machines
    __int64 total_success = 0;
    __int64 total_itr = 1;

    if (first_interpret)
    {
        sim_started = GetTickCount();
        first_interpret = false;
    }
    report *r;
    CSingleLock s(&(pmain->sema));
    s.Lock();
    //get all updates and add total success value
    //and total iterations
    for(int i=0; i < total_nodes; i++)
    {
        r = (report *) pmain->get_update (i);
        total_success += r->success;
        total_itr += r->itr;
    }
    s.Unlock();
    double persec = 0;
    persec = (total_itr - prev_total_itr)/2.0;

    if(persec >= 0)
    {
        prev_total_itr = total_itr;
        double ans = ((total_success * 1.0)/total_itr)*4;
        //print the desired result
        pmain->print("pi = %.10f after %.2f M iter, ||%.2f|| M/sec, run time %d
        sec\n", ans, (double)total_itr/1e6, persec/1e6, (GetTickCount()
        - sim_started)/1000);
    }
}
```

APPENDIX B

0-1 KNAPSACK PROBLEM

The 0-1 knapsack problem is posed as follows. A thief robbing a store finds n items; the *ith* item is worth v_i dollars and weighs w_i pounds, where v_i and w_i are integers. He wants to take as valuable a load as possible, but he can carry at most W pounds in his knapsack for some integer W. Which item should he take? (This is called 0-1 knapsack problem because each item must either be taken or left behind; the thief cannot take a fractional amount of an item or take an item more than once).

Dynamic Programming Solution

Let i be the highest-numbered item in an optimal solution S for W pounds. Then *S'* $= S$ - i is an optimal solution for W - w_i pounds and the value to the solution S is V_i plus the value of the subproblem. We can express this fact in the following formula: define $c[i, w]$ to be the solution for items 1,2, . . . , i and maximum weight w. Then,

$$c[i,w] = \begin{cases} 0 & if i = 0 or w = 0 \\ c[i-1,w] & if w_i \geq 0 \\ max(v_i + c[i-1, w-w_i], c[i-1,w]) & if i > 0 and w \geq w_i \end{cases} \tag{B.1}$$

This says that the value of the solution to i items either include i^{th} item, in which case it is v_i plus a subproblem solution for $(i-1)$ items and the weight excluding w_i, or does not include i^{th} item, in which case it is a subproblem's solution for $(i-1)$ items and the same weight. That is, if the thief picks item i, thief takes v_i value, and thief can choose from items $w - w_i$, and get $c[i-1, w-wi]$ additional value. On other hand, if thief decides not to take item i, thief can choose from item 1,2, . . . , i- 1 up to the weight limit w, and get $c[i-1, w]$ value. The better of these two choices should be made.

The algorithm takes as input the maximum weight W, the number of items n, and the two sequences $v =< v1, v2, ..., vn >$ and $w =< w1, w2, ..., wn >$. It stores the $c[i, j]$ values in the table, that is, a two dimensional array, $c[0..n, 0..w]$ whose entries are computed in a row-major order. That is, the first row of c is filled in from left to right, then the second row, and so on. At the end of the computation, $c[n, w]$ contains the maximum value that can be picked into the knapsack.

```
Dynamic-0-1-knapsack (v, w, n, W)


FOR w = 0 TO W
    DO  c[0, w] = 0
FOR i=1 to n
    DO c[i, 0] = 0
        FOR w=1 TO W
            DO IFf wi = w
                THEN IF  vi + c[i-1, w-wi]
                    THEN c[i, w] = vi + c[i-1, w-wi]
                    ELSE c[i, w] = c[i-1, w]
                ELSE
                    c[i, w] = c[i-1, w]
```

The set of items to take can be deduced from the table, starting at $c[n.w]$ and tracing backwards where the optimal values came from. If $c[i, w] = c[i-1, w]$ item i is not part of the solution, and we are continue tracing with $c[i-1, w]$. Otherwise item i is part of the solution, and we continue tracing with $c[i-1, w-W]$.

Analysis

This dynamic-0-1-knapsack algorithm takes $\theta(nw)$ times, broken up as follows: $\theta(nw)$ times to fill the c table, which has (n+1).(w+1) entries, each requiring $\theta(1)$ time to compute. *O(n)* time to trace the solution, because the tracing process starts in row n of the table and moves up 1 row at each step.

VITA

Kunal S. Patel received his Bachelor of Engineering (B.E.) in information technology from Sardar Patel University, India, in July 2005. He received his Master of Science (M.S.) in computer science at Texas A&M University in December.

He joined the Internet Research Lab at the Computer Science Department at Texas A&M University in August 2006. His research interests include peer-to-peer networks, grid computing and designing large resource sharing distributed systems. He may be contacted at:

Kunal S. Patel

3021 Mauna Loa Ct,

San Jose, CA 95132