Vishwa: A reconfigurable P2P middleware for Grid Computations

M. Venkateswara Reddy, A. Vijay Srinivas, Tarun Gopinath, and D. Janakiram
Distributed & Object Systems Lab,
Dept. of Computer Science & Engg.
Indian Institute of Technology Madras, Chennai, India
{venkatm,avs,tarun,d.janakiram}@cs.iitm.ernet.in
http://dos.iitm.ac.in

Abstract

The abundant computing resources available on the Internet has made grid computing over the Internet a viable solution, to scientific problems. The dynamic nature of the Internet necessitates dynamic reconfigurability of applications to handle failures and varying loads. Most of the existing grid solutions handle reconfigurability to a limited extent. These systems lack appropriate support to handle the failure of key-components, like coordinators, essential for the computational model. We propose a two layered peer-to-peer middleware, Vishwa, to handle reconfiguration of the application in the face of failure and system load. The two-layers, task management layer and reconfiguration layer, are used in conjunction by the applications to adapt and mask node failures. We show that our system is able to handle the failures of the key-components of a computation model. This is demonstrated in the case studies of two computational models, namely bag of tasks *and* connected problems*, with an appropriate example for each.*

1 Introduction

Advancements in networking and cheaper computing technology have enabled the Internet to be used for sharing computation[1], instead of just document sharing. The dynamic nature of the Internet in terms of node/network failures poses challenges for large scale computing. Further, the nodes are autonomic, implying that they may join or leave the system dynamically. Thus, solutions for Internet scale computing requires the use of middleware that ensures dependability of applications in spite of resource/network dynamics. Further, the middleware components themselves must adapt to these dynamics (middleware reconfigurability).

Dependability and reconfigurability becomes more crucial in the case of applications that involve inter-task communication. Scientific problems like finite difference, finite element and finite volume methods require intermediate results to be exchanged, after every iteration. In such connected problems, even if a single node fails or slows down, the entire computation could fail or be stalled. Failures as well as load dynamics dictate that applications need to be dynamically adapted to improve the overall throughput.

Grid computing has emerged as a technology for resource sharing across virtual organizations [5]. However, to the best knowledge of the authors, existing grid technologies provide only limited reconfigurability and scalability. This is mainly due to the presence of dedicated and centralized components, like *GRAM* and *MDS* in the Globus [4] toolkit and, *autonomic problem manager* and *T-space* server in Optimal Grid [8]. Peer-to-peer systems provide solutions to reconfigure from failures and the techniques used in these systems are useful in building scalable systems.

Peer-to-Peer (P2P) systems such as Gnutella [7], Freenet (http://freenet.sourceforge.net), Pastry [11] etc. provide reconfigurability and scalability and are classified into structured and unstructured networks. In unstructured P2P systems such as Gnutella and Freenet, the overlay is built in an uncontrolled fashion. Such systems support arbitrary queries to locate files. In contrast, structured P2P systems assign static identifiers to peers and impose a overlay structure based on the node identifier. The overlay forms a distributed hash table, as in Chord [14] and pastry [11], to provide bounds on query times.

However, these file sharing P2P systems may not be directly usable for sharing compute resources on the Internet. This is because these systems only provide a routing mechanism without any additional support for handling computations. Efforts, like [1, 3, 13] make use of such peer-to-peer routing techniques to support computation over the Internet. These systems support only limited reconfiguration. In order to provide reconfiguration even in the case of failures of certain key functional components, like coordinators, which are essential to the computational models used over grids,

we propose a two-layered P2P architecture, Vishwa.

The remainder of the paper is organized as follows. Section 2 explains the the middleware and its components. Section 3 evaluates the system interms of two grid-computational class of applications. Section 4 provides our analysis of the scalability of Vishwa. Section 5 compares our model with other similar efforts. Section 6 provides our concluding remarks on this system.

2 Vishwa: A Two Layered P2P Middleware for Grid Computing

2.1 Two Layered Architecture

The two layered architecture at the heart of our middleware, provides applications with the ability to progress, inspite of failures to nodes and key components. This section describes the functionality of the two layers.

The task management layer provides proximity based clustering of resources. In other words, the task management layer groups the nodes of the system into proximity based zones or clusters. Within a zone, nodes maintain neighbour lists. The nodes add other nodes with higher Horse Power Factor (HPF) as neighbours. The HPF [12] is a measure of the capability of a node, in terms of computation, memory and communication. The neighbour list of a node is altered dynamically, since HPF is a dynamic factor and failures may occur in the system. The task management layer implements a HPF propagation algorithm, which propagates dynamic HPF values across nodes. It also implements a heart beat algorithm to detect node failures. Both the above algorithms result in the neighbour lists being updated across nodes. Details of the HPF propagation algorithm and the heart beat algorithm will be discussed in later sections.

The reconfiguration layer enables Vishwa to handle node/network failures. The information required to recover from failures, such as task details (task id, executing node etc.) and intermediate results are stored in this layer. Each piece of information gets a quasi-random identifier and is replicated in k other nodes of the system. The structured P2P layer ensures that if even one of the k nodes is alive, the information can be recovered in O(log(n)). The k replicas are maintained in the same cluster/zone. This makes it easier to maintain consistency among the k-replicas[1].

Figure 1 gives the block diagram of components in each node of the Vishwa middleware. The following sections will bring out the details of these components, with the bootstrapping component being explained in the next section.

[1]This is one reason why proximity based zones are maintained. The other reason is that computing elements can be clustered closer, resulting in faster communication between the computing elements.

2.2 Grid Initialization: Bootstrapping

When a node joins the grid, it contacts the closest available zonal server[2]. The zonal server, when contacted by a grid node for joining, returns a list of neighbours based on proximity. The metric used for proximity is the hop count. The joinee can choose its neighbours from among this list based on the network delay. The zonal server also returns a list of proximally closer zonal servers to the joinee. This is to ensure that the joinee gets some neighbours from other zones, in order that zones are not completely isolated from each other.

The Distributed Hash Table (DHT) used by the reconfiguration layer, is also constructed at grid initialization time. The algorithm for routing state [3] initialization is explained in [9].

2.3 User Daemon Component: Initial Task Submission

The user daemon component of Vishwa is responsible for interaction between the end user and grid. The end user submits the task and gives the task configuration details. The user daemon process generates a task identifier by using the Id_Generator module of Vishwa[4]. The Task-Id is used to route the task to the node with the closest matching Node-Id. This node acts as a coordinator for this task.

2.4 Problem Manager Component: Task Splitting

The node which has been chosen as the coordinator for a particular task runs the problem manager component of Vishwa. The problem manager component requests the scheduler for the required number of donors (based on the user input) by supplying it with the task type. The scheduler returns the list of donors along with their HPF values. Based on the number of donors available and their HPF values, the problem manager splits the problem into chunks. It sends data and sub-tasks to the dispatcher for dispatching to appropriate grid nodes.

The problem manager also stores information about tasks, such as Task-Id, submitting node, donors and sub-task information in the reconfiguration layer. This information is useful for reconfiguration in the case of failures.

[2]Zonal server generates node identifiers and enables grid nodes to know each other and form clusters.

[3]Routing state of a node contains a *leaf set and a routing table*

[4]The task identifier contains the zone identifier, as explained the Route Manager component

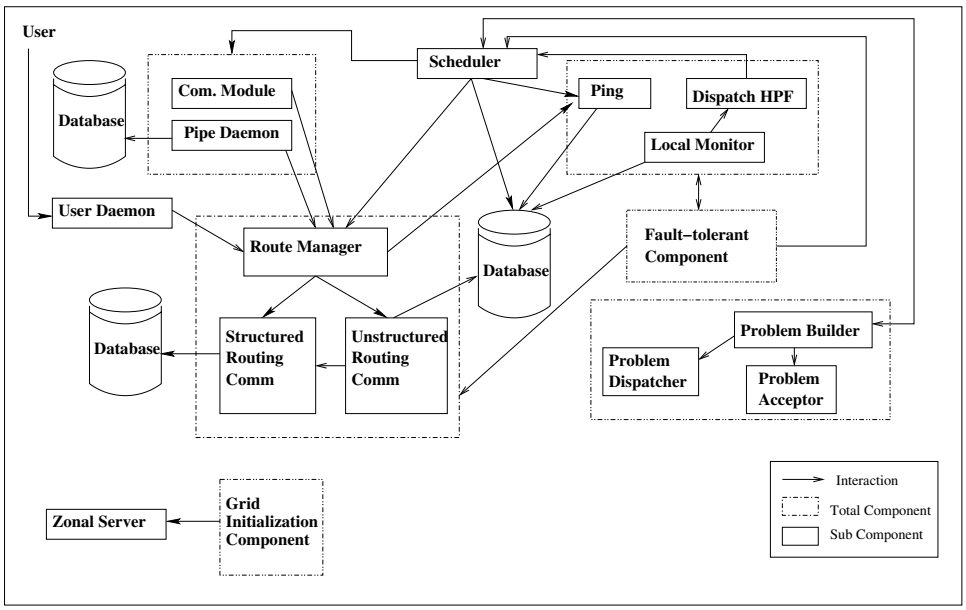

Figure 1. Vishwa: Node Level Block Diagram

2.5 Sub-task Executor Component

The sub-task executor component in each node is responsible for executing the given sub-task locally. It stores information about the sub-task, including sub-task identifier, coordinator for the sub-task and task identifier in the reconfiguration layer with the sub-task identifier as the key. In the case of unconnected problems, it also sends a message containing sub-task id and coordinator details to the monitoring component of neighbour nodes in the task management layer. This enables the detection of node failures. In the case of connected problems, it sends the same message to the monitoring component of application level neighbours. Application level neighbours are those that communicate intermediate results after each iteration in connected problems. It also checkpoints the intermediate results and stores them in the reconfiguration layer with sub-task-id as key. It is also responsible for sending the final results of the sub-task to the coordinator.

2.6 Scheduler Component

Vishwa uses a fully decentralized scheduler. The scheduler looks at the neighbour list to obtain the list of nodes available within k-hops. This gives the list of nodes to which tasks can be migrated from this node.

HPF is primarily used to select candidate nodes to which tasks can be migrated (nodes with high HPF can be choosen). Nodes advertise the HPF values to neighbours using the task management layer. This advertising of HPF value of a node cannot be adhoc. An adhoc dissemination of information may result in a node with high HPF never being utilized. To prevent this from happening following heuristic is used.

1. advertise HPF value within a logical hop radius of k; where k is either a prespecified system parameter or a parameter that is calculated based on initial configuration or overlay topology.

2. For each node j that is its logical neighbor, node i checks if HPF_i is greater than HPF_j and if so, then the HPF_i value is propagated to node j. Node j might propagate it further to its neighbours with HPF values lesser than HPF_i.

3. The above process is carried out till a maximum of k logical hops.

The scheduler uses both sender-initiated and receiver-initiated schemes for task migration. Sender-initiated implies that if HPF of a node increases above or below a given threshold, the HPF propagation algorithm is invoked. Receiver-initiated scheme implies that whenever a node needs to send sub-tasks to nodes and it does not have sufficient number of nodes in its neighbour list, it can use an expanding ring algorithm to get the donors.

2.7 Monitoring Component

The monitoring component of Vishwa is used by the middleware to keep track of resource dynamics of the grid and is completely transparent to applications. The main

functionalities of this component include monitoring load conditions in its grid node and keeping track of the failure of other related grid nodes. If the HPF value crosses lower or upper thresholds, it informs the scheduler which in turn propagates the HPF value to other nodes.

The monitoring component also monitors the failure of related grid nodes. By related, we mean three kinds of nodes: neighbour set nodes of the task management layer, leaf set nodes[5] in the reconfiguration layer, application level neighbours (in case of connected problems). The monitoring component uses a heartbeat algorithm to monitor related nodes. It sends an application level ping message to related nodes. The nodes respond with their HPF value. The monitoring component calculates the delay based on the response time. If a node does not respond for a predefined number of ping messages, the monitoring component assumes that the node has failed. It informs the same to the fault-tolerance component.

2.8 Route Manager

The main functionalities of the route manager is to store/retrieve information from the reconfiguration layer and to route messages. In the task management layer, the scheduler or dispatcher (to send messages to neighbours) interacts with the route manager to get the neighbour list of a node. This list is updated by the monitoring component, as explained earlier. The route manager also routes messages in the reconfiguration layer, details of which are explained below.

The reconfiguration layer is designed as a structured P2P layer. The intra-zonal routing is similar to Pastry [11], while the inter-zonal routing imbibes concepts similar to Chord. The reconfiguration layer guarantees that information can be retrieved in $O(log(N))$ bound, for N nodes in the system. The node identifier generation process is different in Vishwa compared to the purely random generation of Pastry. The zonal identifier, a logical number that represents the zone, is appended to the hashed IP address of the node to get the node identifier. The motivation behind this process of ID generation is to ensure that replicas that store reconfiguration information are within the same zone in order to make it easier to keep them consistent.

The inter-zonal routing can be explained as follows: To route to a node across zones, the zone id is masked. The message is routed to closest matching node within the same zone. If the closest matching node already has an entry for the destination zone, it directly forward the message to the corresponding node in the destination zone. If the closest matching node does not have an entry for the destination zone, it routes the message to the zone at distance less than 2^m and is closer to the destination zone, whose entry exists. The same steps are repeated at intermediate zones, till the destination zone is reached. The complete inter-zonal and intra-zonal routing algorithms can be found in [9].

[5] In Pastry, leaf set nodes refer to nodes that are closer in the Node-Id space to a given node.

2.9 Communication Component

The functionality of the communication component is to provide support for inter-task communication. Inter-task communication is essential for connected problems, such as iterative grid/mesh computing, Pizo-electrical coupled problems, Electro-magnetic coupled problems, Thermo-Elastic problems, Active Vibration problems, etc. Vishwa provides support for inter-task communication building on the Distributed Pipes (DP) [6] abstraction. DP provides the abstraction of pipes to facilitate connected problems. Vishwa stores pipe information including pipe identifier and read/write *ends* in the reconfiguration layer. This implies that in case of failures, pipe information can be retrieved within $O(log(N))$ even across zones, enabling connected problems to be executed on grids.

3 Case Studies

We study two commonly occurring class of applications, namely *Bag of Tasks* and *Connected Problems*[8] and the support for their computational models over our middleware. The computational models used by these classes of applications are explained along with performance evaluation for particular instances in the following subsections.

The computation in our grid middleware is initiated by the user on the user-node, who submits a grid task to one of the grid nodes along with the task configuration details. The configuration details include, the number of donors required, type of task and a meta-file containing task input descriptions. The middleware determines the coordinator for the task based on the task-id and then sends the task along with the configuration details to the coordinator, through the reconfiguration layer. The coordinator utilizes this information to split the tasks appropriately and distribute them over available nodes. The model of computation to be used for a given application is determined from the user-submitted type information.

The instances of the computational model are studied over an intranet and an Internet testbed. The intranet testbed consists of 35 nodes within the institute, with nodes having memory from 64MB to 1GB, processing speed from 350MHz to 2.4GHz. The Internet testbed consists of nodes from MAM college of engineering and from University of Melbourne, Australia.

3.1 Case Study: Bag of Tasks

3.1.1 Execution Model

The *problem manager* component of the coordinator, to which the application is submitted, requests the *scheduler* for the required number of donors. The scheduler returns a list of donors along with their HPF values. Based on the number of donors available and their HPF values, the *problem manager* splits the problem into subtasks. These subtasks are dispatched to the appropriate grid nodes. Information about the task, subtask and the nodes on which they are executing is maintained in the reconfiguration layer.

The individual nodes on which the subtasks execute, register with neighboring nodes (neighbor set) in the task management layer. Intermediate results of the subtasks are checkpointed and saved in the reconfiguration layer to ensure recovery from failures. If the load on the sub-task-node increases beyond a threshold, the monitoring component notifies the scheduler to initiate task migration.

The failure of the sub-task-node is detected by the neighbor set, which inturn informs the coordinator through the reconfiguration layer. The coordinator retrieves the checkpointed state of the sub-task from the reconfiguration layer and selects another node on which the sub-task is restored. The coordinator updates the subtask information in the reconfiguration layer appropriately.

The coordinator failure can occur either, before the initial task is submitted to it, from the user-node or after subtasks are dispatched. In the former case, the user-node resubmits the task, while in the latter case the reconfiguration layer ensures that information directed to the coordinator is redirected to the node with the next closest id matching the task-id (similar to Pastry[11]). This node then takes on the role of the coordinator for the specified task-id.

An example application which uses the bag of task computational model is the neutron shielding application. We evaluate the performance of this application over our middleware.

3.1.2 Neutron Shielding Application

Neutron Shielding is a nuclear physics application, used to determine the number of neutrons penetrating through a given thickness of lead sheet. The experiment predicts this number using Monte-Carlo simulation. In our experiments we assume a lead sheet of 5 units thickness and vary the number of neutrons in powers of 10. This is a compute intensive application.

Figure 2 shows the execution time for increasing number of nodes for the neutron shielding application. The first graph shows the results of the application execution under no-load and heavily loaded conditions. The neutron shielding application is able to adapt to the dynamic load fluctuations, due to Vishwa. The graph shows that the overhead of load balancing is small on the overall computation time. The results show that Vishwa can achieve linear speedup in a loaded cluster. Speedup is defined as the ratio of parallel execution time of the problem to its sequential execution time.

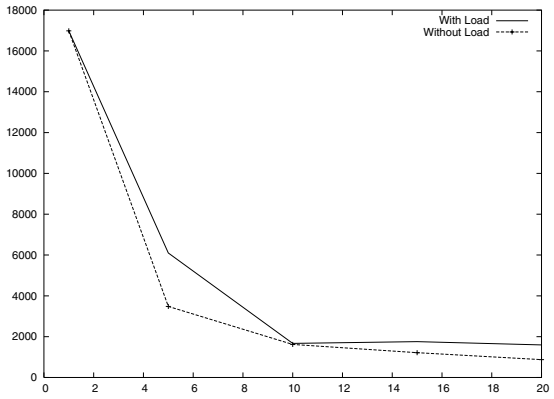

Figure 2. Execution time VS number of nodes: no load and loaded conditions

Executing the same application on loaded nodes without task management shows that the performance degradation is large compared to application execution over Vishwa. This is evident in table 1.

Table 1. Performance of Neutron Shielding Simulation on a Loaded Cluster

No. of Nodes	Execution Time Without Load Balancing	Execution Time with Vishwa Load Balancing
	(Time in seconds)	
4	8561	4345
15	1756	1129
20	2159	1074

The overhead of *reconfigurability* for this application in the case of sub-task-node failure and coordinator failure is presented in the tables 2 and table 3, respectively.

Table 2. Fault-tolerance Overheads: Node Failure in Unconnected Problem

Registration Overhead	Fault Detection Overhead	Task Reconfiguration Overhead
(Time in seconds)		
0.525	0.212	11.131

The overhead of storing information in the reconfiguration layer ($k - replicas$) was measured and found to take about 1.3 secs on average (it must be noted that the replicas will be within the same cluster). This overhead was nearly a constant and in the order of O(log(N)), here N is number of machines, as shown in table 3. This implies that with minimal overhead, the coordinator failure can be masked to the user process.

Table 3. Overhead of Storing and Retrieving in Reconfiguration Layer

No. of Nodes	Overhead (Time in seconds)
4	1.32
8	1.29
12	1.31

3.2 Case Study: Connected Problem

3.2.1 Execution Model

Vishwa provides support for inter-task communication, building on the distributed-pipes[6] abstraction. In this execution model, the *problem manager* on the coordinator, requests the *scheduler* for a given number of donors. The split tasks are dispatched to the nodes along with the end points of the communicating tasks. The coordinator, then stores pipe information and the task information in the reconfiguration layer.

The sub-task-nodes begin the execution of their respective subtasks, after establishing pipes between themselves. The sub-tasks register with the nodes with which pipes are established, in order to handle failure. The sub-tasks checkpoint intermediate results, similar to the bag of tasks computational model, described in section 3.1.

Whenever a node running a sub-task, detects the failure of the other end of the pipe, it notifies the coordinator through the reconfiguration layer. The coordinator, then stops all the individual sub-tasks by sending appropriate signals to the sub-task-nodes. The node which detected the failure in the first place (failed-pipe-node), then chooses a new node to restore the failed sub-task on. The new node is selected from the task-management layer. The failed-pipe-node, then informs the coordinator of the selected node. The coordinator updates this information in its base and restores the computation by sending a signal to each of the sub-task-nodes.

Dynamic load balancing in these applications is handled similar to the case of the sub-task failure. The only difference is that in this case the node which requires to balance the load initiates the process instead of the failed-pipe-node. Coordinator failures are handled similar to that described in section 3.1.

Steady State Equilibrium Application is an example of such a connected problem. The performance of this application is evaluated in the following section.

3.2.2 Steady State Equilibrium Application

The Steady State Equilibrium Problem computes the temperature distribution of a rod whose ends are kept at fixed temperature baths. This is a fluid dynamics problem[6]. The problem iteratively computes the temperature values at equally spaced grid points on the rod, at regular intervals of time. Details of this application can be found in [6].

Super-linear computational speedup was achieved for the Steady State problem, as depicted in the graph in figure 3. The speedup is attributed to the parallelism and reduced memory requirements on individual nodes. These super linear speedups can be achieved as long as the problem runs on optimal number of nodes with appropriate grain size. If the grain size of the sub domain is small, the communication overhead increases causing decrease in speedup.

The grain size of a subtask is the number of slice points alloted to a subtask. Task time of a subtask is the time taken for computation of the subtask which is the sum of actual CPU time of the subtask and synchronization delay suffered by subtask.

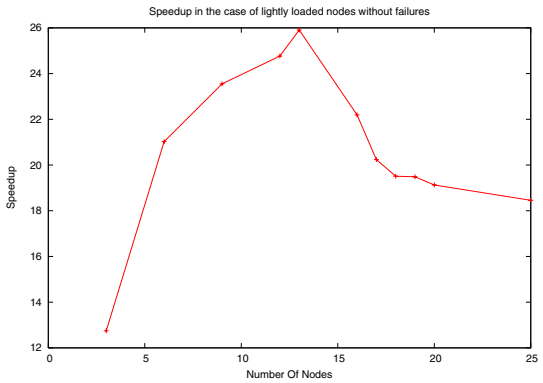

Figure 3. Speed up in the case of lightly loaded nodes without failures

The prototype results as shown in table 4 have shown that the speedup increases super-linearly as we increase the number of nodes. But this increase is limited up to some extent. The speedup increased when the number of nodes increased to 13. But the speedup decreased as the number of nodes was increased above 13 and remained almost constant showing a saturation point. This was observed when the number of nodes was more than 18.

Table 4. Speed up in the case of lightly loaded nodes with out failures

No. of Nodes	Grain Size	Total Task Time (secs)	Speedup
1	10,00,00,000	1489.621789	-
3	3,30,00,000	116.876431	12.745273
6	1,62,50,000	70.856432	21.023099
9	1,11,11,000	63.254592	23.549623
12	83,33,000	60.149577	24.765291
16	62,50,000	67.106057	22.198023
18	55,55,555	76.353495	19.509543
20	50,00,000	77.887006	19.125421
25	40,00,000	80.708851	18.456734

When the same computing problem with same grain size was executed on the WAN test bed, more synchronization delay between subtasks was observed. This is mainly due to communication latencies. When the computation is run on the Institute test bed, higher speedups are obtained. However, even in the wide area testbed, we were able to obtain super linear speedup for sufficiently large problem sizes. The graph in figure 4 shows this case.

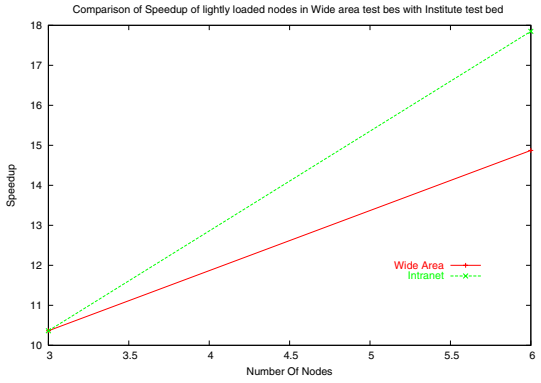

Figure 4. Comparison of Speedup in Intranet and Internet Test-beds

Table 5 shows the overhead of grid initialization, with increasing number of nodes. The fourth column also shows the resource discovery overheads. It can be observed that the grid initialization and resource discovery overheads are nearly constant with increasing number of nodes, implying that the middleware is expected to scale well. Moreover, these overheads do not depend on the computational model.

Table 5. Overhead of Grid Initialization with increasing nodes

Number of Nodes	Registration (secs)	Neighbour List Updation (secs)	HPF Propagation (secs)
1	0.862	0.000032	0.000003
5	0.848	5.002	0.328
12	0.885	10.011	0.399
18	1.167	10.012	0.352
20	1.163	10.011	0.474
23	1.142	10.01	0.442

4 Related Work

In this section we compare Vishwa with the existing grid middleware along different dimensions. The traditional grid middlewares like Globus [4], gridbus (http://gridbus.org), Optimalgrid [8] etc. have centralized components like administration and management components. These components could become bottlenecks, and single points of failure, as the scale increases. They use client server paradigm to build the grid. Thus, the existing grid middlewares may not scale. They do not support reconfigurability in the case of failures.

4.1 Comparison with Internet Scale Computing Efforts

SETI[1], distributed.net (http://www.distributed.net/) Folding@home (http://folding.stanford.edu) harness the idle computing power on the Internet. These are desktop grids which have been scaled up to the Internet. However, these are restricted to a specific application. The underlying architecture is more like a *star*. This implies that centralized servers exist, leading to single points of failure.

Application dependability and middleware reconfigurability are important issues if we aim for Internet scale computing. Recently, an attempt for application reconfigurability to balance the load among MPI processes of geographically distributed nodes has been made [10]. But they did not provide any support for application reconfigurability in the case of computational node failures. In contrast, Vishwa supports application reconfigurability in the case of node and even coordinator failures.

4.2 Grids and P2P Integration Systems

P3 [13] is a P2P middleware to enable transfer and aggregation of computing resources. It uses Juxtapose (JXTA)

as the base P2P library. It is important to note that P3 uses only an unstructured P2P overlay. This implies that in the case of failures, information retrieval (to allow reconfiguration) cannot not be guaranteed nor bounded. In [13], the authors do not handle failures of the controller or even compute nodes. Ourgrid [3] is a Peer to Peer middleware to share computing cycles across organizations. The main motivation of ourgrid project is to develop a middleware which automatically collect the resources across several organizations and provide easy access to the resources. It also developed a network of favour economic model to avoid the free riding problem among the peers. The middleware well suited to solve Bag of Task applications in the failure free environment. It doesn't provide support for iterative class of applications.

Self organizing flock of Condors[2] is a self organized fault tolerant resource discovery mechanism, which uses peer-to-peer mechanisms. It does not provide an execution environment to run the applications. It aids to build the execution environment.

5 Conclusions

We have proposed a unique two layered P2P middleware for Internet distributed computing that is available for download from http://dos.iitm.ac.in/Vishwa. The middleware supports application and middleware reconfigurability by utilizing the proposed two layered architecture. The two layers leverage the utility of unstructured P2P and structured P2P in order to achieve this form of reconfigurability in case of failures. The structured layer reconfigures the application to mask failures, while the unstructured layer reconfigures the application by adapting to the varying loads. The results obtained encourage the use of such a two layered architecture to build reconfigurable computational models over the Internet. The case studies demonstrating the use of the architecture for two different classes of applications, provides a substantiative argument in favour of the system's use. Future research directions include exploring the feasibility of using Vishwa for building a data grid middleware.

Acknowledgments
The authors thank the members of the DOS Lab for their useful comments. We also acknowledge the support of the Department of Science & Technology (DST) for supporting part of the research project.

References

[1] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@home: An Experiment in Public-Resource Computing. *Communications of the ACM*, 45(11):56–61, November 2002.

[2] A. R. Butt, R. Zhang, and Y. C. Hu. A self-organizing flock of condors. In *SC*, page 42. ACM, 2003.

[3] W. Cirne, D. P. da Silva, L. Costa, E. Santos-Neto, F. V. Brasileiro, J. P. Sauvé, F. A. B. Silva, C. O. Barros, and C. Silveira. Running bag-of-tasks applications on computational grids: The mygrid approach. In *ICPP*, pages 407–. IEEE Computer Society, 2003.

[4] C. K. Foster. Globus: A metacomputing infrastructure toolkit. *Intl J. Supercomputer Applications*, 11(2):115–128, Fall 1997.

[5] I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal on Supercomputer Applications*, 15(3), 2001.

[6] B. K. Johnson, R. Karthikeyan, and D. J. Ram. Dp: A paradigm for anonymous remote computation and communication for cluster computing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1052–1065, 2001.

[7] P. Karbhari, M. Ammar, A. Dhamdhere, H. Raj, G. Riley, and E. Zegura. Bootstrapping in Gnutella: A Measurement Study. In *Proceedings of Passive and Active Measurement Workshop PAM 2004*, Antibes Juan-les-Pins, France, April 2004. Springer in the Lecture Notes in Computer Science (LNCS) series.

[8] T. J. Lehman and J. H. kaufman. Optimal Grid: Middleware for Automatic Deployment of Distributed FEM Problems on an Internet-Based Computing Grid. In *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER'03)*, 2003.

[9] M Venkateswara Reddy, A Vijay Srinivas, Tarun Gopinath, and D. Janakiram. Vishwa: A Scalable Reconfigurable P2P Middleware for Grid Computing. Technical Report IITM-CSE-DOS-2005-12, DOS Lab, Department of CS&E, Indian Institute of Technology Madras, 2005. Available from http://dos.iitm.ac.in.

[10] K. E. Maghraoui, T. Desell, B. K. Szymanski, J. D. Teresco, and C. A. Varela. Towards a middleware framework for dynamically reconfigurable scientific computing. In L. Grandinetti, editor, *Grid Computing and New Frontiers of High Performance Processing*. Elsevier, 2005. to appear.

[11] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Midleware 2001)*, pages 329–350, Heidelberg, Germany, Nov 2001.

[12] D. J. Rushikesh K. Joshi. Anonymous remote computing: A paradigm for parallel programming on interconnected workstations. *IEEE Trans. on Software Engineering*, 25(1):75–90, Jan/Feb 1999.

[13] K. Shudo, Y. Tanaka, and S. Sekiguchi. P3: P2P-based Middleware Enabling Transfer and Aggregation of Computational Resources. In *Proceedings of the Fifth International Workshop on Global and Peer-2-Peer Computing*, Cardiff, UK, 2005.

[14] Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *Proceedings of ACM SIGCOMM, San Diego*, pages 160–177, Aug 2001.