Massively Parallel Processing on Peer-to-Peer Systems

Jon Ludwig, Prashant Gahlowt, and Young Suk Moon

Department of Computer Science
Rochester Institute of Technology
{jcl1827, pxg3428, yxm9371}@cs.rit.edu

Contents

1 Introduction

With the increase in the complexity of the nature of distributed tasks several systems face challenges in providing sufficient amount of resources on one system to carry out the task efficiently. On the other hand failure in processing of such a complex jobs could result in huge amount of wastage, both in terms of resources as well as time. Our system presents one such approach to carry out efficiently similar kind of massive computing job. To build our system we have chosen the base network model to be Pastry, a scalable, distributed object location and routing substrate for peer-to-peer applications over the Internet. The application that we chose to show the working and effectiveness of our system is of distributed rendering of randomized fractal images. The system we have developed is characterized by some very interesting qualities of being completely decentralized, self-organized, adaptable, scalable and fault-resilient. Throughout the paper we would substantiate our idea by a series of design details and experimental evaluations.

The traditional client/server architecture is not a good fit for a massively large and resource consuming job due the limitations in resources that a single machine can devote. To achieve such a large amount of computational resource, our system incorporated a decentralized peer-to-peer architecture called Pastry. This architecture gave us a aggregation of large amount of computers along with their resources which made the structure more feasible for carrying out these massive tasks. On top of Pastry, we built a work-distribution model that granulated this massive job into several smaller jobs and distributed it over the network for multiple computers to work on their assigned piece. Since the approach was highly parallelized and the serial computational requirements were minimal, Pastry qualified to be a suitable match for our application.

The job of distributed rendering of fractal images is capable of being efficiently divided into an arbitrary number of smaller jobs. These sub-jobs are independent of each other and can be successfully carried out on isolated machines without any feedback required from any other sub-jobs. Also another characteristics of the system is that each node participating in computation was interested in the finished product once all the computation has been carried out. How we carried out this complex task is by following a prototype where the node which is assigned a job is designated as a master node, this master node then renders slices of fractal images in parallel over a number of worker nodes.

Pastry formed the basic building block of our network model. Its inherent properties of decentralized control, self organization, adaptability, scalability and fault resilience provided our system a great amount of flexibility. The decentralized control of Pastry equipped us with providing the nodes equal opportunity and capability while processing jobs. Pastry handles the joining and leaving of nodes in a very graceful manner. All the concerned nodes are informed of a change in technology. Its capability to reconstruct the network in case of change in network also turned out to be very helpful for our application. The paper we analyzed on Scribe helped us in supporting our master-worker model for work distribution. The idea we implemented from Scribe was in form of Topic subscrip-

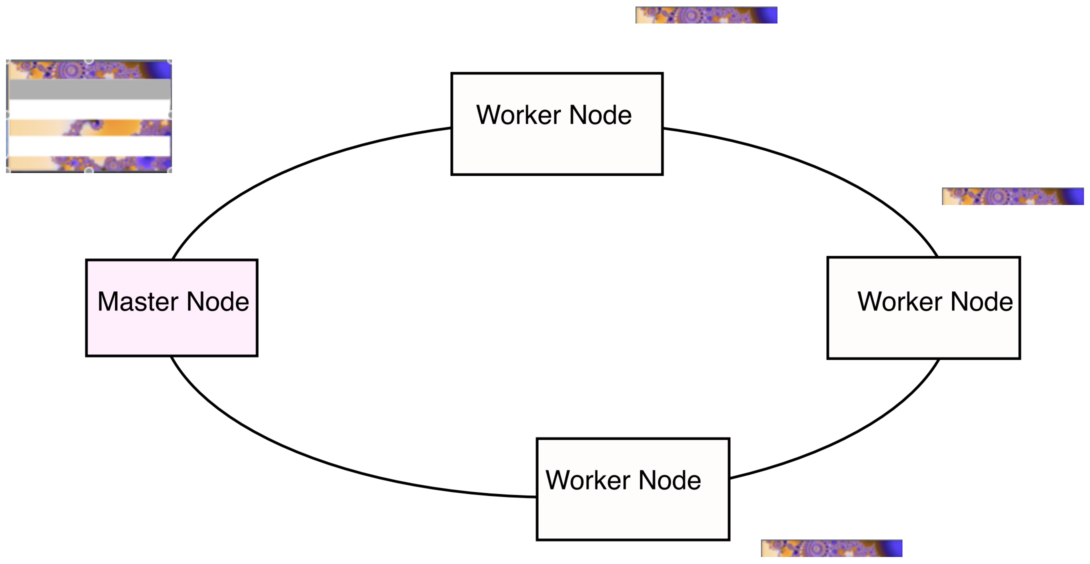


Figure 1: The work distribution amongst the Master and the Worker nodes.

tion. In Scribe a Topic can be published, the node that publishes the topic is assigned as the master node, all nodes that are interested in this topic could subscribe to it and act as worker nodes. Later the master node could communicate with these worker nodes by the use of multicast or anycast. In the design part of the paper, we will get into further details of how we benefited from the use of Pastry and Scribe.

2 Previous Work

In this section three works that were researched in detail are discussed. The first paper [9] describes the peer-to-peer overlay system which was used to maintain connections and route messages between node at the application level. In [4] a multicast extension to the peer-to-peer overlay is described. The third paper [5] describes load balancing strategies which influenced our own implementation.

2.1 Peer-to-Peer Overlays

With the advent of the Internet, the domain of network applications have had an immense growth both geographically as well as in terms of complexity. Unlike in the case of traditional client/server architecture, these applications require a far more scalable and robust infrastructure. Peer-to-peer systems are one such approach that show the potential to encompass the growing needs of these large networks. This architecture is characterized by decentralized control, self-organization, adaptation and scalability. This paper describes one such architecture called Pastry [9, 3]. Pastry is a scalable distributed object location and routing application for wide area peer-to-peer applications. It is completely decentralized, scalable, self-organizing and fault resilient. All the nodes participating in these systems have symmetric capabilities and responsibilities. These properties allow Pastry to support a large number of nodes using its application level routing over the internet for global data storage, data sharing, group communication and naming. The paper discusses in detail the design and evaluation of Pastry and proposes it to be a good solution addressing the current network infrastructure demands.

Each node in Pastry is uniquely identified with a 128-bit *nodeId*. This *nodeId* symbolizes the position of this node in the Pastry ring, which ranges from 0 to $2^{128} - 1$. These *nodeId*s are assigned randomly and thereby assure a uniform distribution over the ring. These nodeIds are generated using a cryptographic hash of a nodes public key or its IP address. This assure that each node has a unique Id and there are no duplicate nodes. When a message arrives at a node in the Pastry ring, the message is efficiently routed to the alive node with the nodeId that is numerically closest to the key of the message. Assuming a ring with N nodes, under normal conditions Pastry assures a message route in less than $\lceil \log_{2^b} N \rceil$ steps, where b is a configuration parameter with a typical value of 4. In adverse conditions where nodes fail continuously, Pastry guarantees the message delivery unless adjacent $\lceil |L|/2 \rceil$ nodes fail simultaneously, $|L|$ is a configuration parameter with a typical value of 16 or 32.

Pastry maintains three data structure, to support its message routing, routing table (R), neighborhood set (M) and leaf set (L). Routing table consists of $\lceil \log_{2^b} N \rceil$ rows with each row containing $2^b - 1$ entries. Each entry in the row of the routing table refers to a node whose nodeId shares the present nodes nodeId in the first n digits, but the $n + 1$th digit has one of the $2^b - 1$ remaining values other than the $n + 1$th digit in the nodes Id. The neighborhood set M contains IP addresses of nodes that are closest to the current node according to the proximity metric. These entries are not used in routing, they help in maintaining locality properties discussed in the later part of the paper. The leaf set L has the entries of the nodes with $|L|/2$ closest smaller nodeIds and $|L|/2$ closest larger nodeIds. These entries are used while routing messages. The typical values of $|L|$ and $|M|$ are assumed to be 2^b or 2×2^b.

Pastry routing mechanism triggers when a message arrive at a node. The node that receives this message first checks its leaf set to see if the message key falls in that list. If a match is found or a node with nodeId that is the closest match to the message key, the message is directly forwarded to the destination node. Incase if no appropriate match is found in the leafset, routing table is used to find a node with the nodeId that shares a common prefix with the key and matches at least one more digit. In certain rare cases this entry could be found to be empty, in which case the message is forwarded to a node that shares the same prefix match as the current node but is numerically closer to the key than the present node. In an event vent were many simultaneous nodes fail simultaneously, the message delivery time get linear in N. In the worst case where $\lceil |L|/2 \rceil$ adjacent nodes fail simultaneously, the message delivery can not be guaranteed. The process of routing always converges as each step takes the message to a node that either has a longer prefix than the current node or with the same prefix match but is numerically closer to the key than the local node.

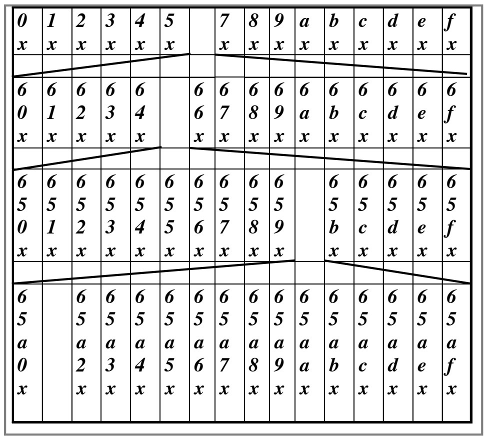
One of the main inherent properties of Pastry that the paper describes is of self-organization and adaptation. Pastry nodes keep a track of their neighboring nodes in a node space, this helps in alerting the nodes in case of change in topology of the ring. Pastry handles the arrival and departure of nodes very gracefully. When a node arrives in a pastry ring, it builds its state table and then informs the concerned nodes about its presence. The node that intends to join an existing sends a join message to a randomly selected live node. This node then acts as a bootstrap node for this new node. If the node is the first node, it bootstraps itself and starts a new ring. Once the node has chosen a bootstrap node, the further process is similar to that of a message passing. The bootstrap node passes this new node to the node that has its Id closest to that of the new node. During the process of reaching the appropriate position in the ring, the new node request state information from each node that it hops and uses it to build its state information. Once the new node reaches the appropriate position in the ring, it broadcast its presence to the nodes that need to be aware of its arrival.

The nodes in a Pastry ring tend to fail without any warning message. This failure is detected when a nearby node tries to contact this failed node and gets no response. Once this communication failure has been detected, this neighboring node contacts the other live nodes in the node space. While contacting these nodes, the neighboring node chooses the nodes with the largest index on the side of the failed node and request these nodes for their leaf table. The leafset of this contacted node partly overlaps with the leafset of the present node and some new nodes that are not present in the leafset of the present node. An appropriate nodeId is chosen from this list on non-overlapping nodes and a contact is established to assure that this new node is alive. Later this node builds its leafset lazily and also informs other nodes that have the failed nodes Id in their routing table. A replacement node in the leafset helps in preventing the integrity of the routing table.

Another such property that makes Pastry stand apart from other peer-to-peer architecture like Chord and Can is its ability to maintain node locality information. This property ensure that the route chosen for a message is likely to be a good one with respect to the proximity metrics. The proximity metric in the Pastry architecture is built based upon some scalar proximity metric like number of hops or geographical distance. To build the proximity metrics, Pastry relies on the application. The application built upon Pastry is expected to provide it with its choice of proximity metric. As discussed earlier the proximity metric comes into play from the start when a node requests to join a ring. To preserve the proximity properties a joining node is more likely to contact a bootstrap node that is closer to it. This property is further preserved even when nodes leave the network unexpectedly. Several applications built on Pastry use replication to provide some level of fault tolerance. These nodes that are chosen to store the replicated information are the ones that are closer to the current node in regards to the proximity metric. This ensures that a routed message with the given key is more likely to reach one of these replicated nodes that is still alive.The authors of the paper further support their proposed idea by conducting a series of experiments that puts to test the properties claimed by Pastry. The results show that Pastry performs very consistently in keeping the message routing time to $\lceil \log_{2^b} N \rceil$ even when the number of nodes vary from 1000 to 10,000. In another such setup the route length is again shown to be $\lceil \log_{2^b} N \rceil$ measure in terms of number of hops. Other experiments show the resurgence of the ring in case of consecutive node failures.

The authors propose a very promising architecture in the field of peer-to-peer networking. Use of Pastry as a peer-to peer routing scheme over the Internet promises to be a very effective scheme to address several issues. The inherent properties of Pastry to be completely decentralized, fault-resilient, scalable and reliable in delivering the messages makes it a building block for a variety of peer-to-peer applications over the Internet. Pastry takes into account the locality properties of the ring and guarantees the delivery of messages in a consistent amount of time under regular conditions. The self-organizing characteristics of Pastry and the way it gracefully handle the arrival and departure of nodes make it very appropriate for the ad-hoc dynamic networks. However there are certain questions that still remain unanswered. The neighborhood set present in the node state table has not been shown play much important role. This neighborhood set is not used in the routing the message, however its size and the time and resources it takes to populate it is quite considerable. The authors are expected to reveal further details regarding the use of neighborhood set. Also unlike a very similar architecture, Chord, Pastry does not has the functionality for key migration when a new node joins. This could put limitations to applications that do not incorporate non-replication file sharing.

After considering all the pros and cons of the Pastry architecture, it turned out to be a very obvious choice for our massive parallel computing fractal image generation application. Our application required a dynamic topology in which nodes join and leave the ring very frequently, Pastrys inherent property of adapting to this change in topology gave us a strong foundation to start our work. We used the Pastrys locality property to implement significant level of fault tolerance by the use of replication. In our application a node replicates it completed job to all the nodes in its leafset so as to keep this data available on other nodes even if some of the nodes fail. Another very important property of Pastry that we exploited was that of self-organization. In a Pastry ring when a nodes drops out, all the concerned nodes are informed about this loss. We used this property to reassign incomplete jobs because of node failures. This property was also exploited in our application for role migration, incase of the failure of a master node, the worker nodes were informed about it, they then conducted an election and a new master node is assigned from the set of worker nodes. However along with the features that we found supportive for our application, there were some limitations that were not so suitable. One such limitation was the latency found in the delivery of messages, significantly in case of node drop outs. The time taken for the neighboring nodes to receive the failure of node in a ring was quite significant. This restricted our application to be not so real-time.



0x	1x	2x	3x	4x	5x		7x	8x	9x	ax	bx	cx	dx	ex	fx
60x	61x	62x	63x	64x		66x	67x	68x	69x	6ax	6bx	6cx	6dx	6ex	6fx
650x	651x	652x	653x	654x	655x	656x	657x	658x	659x		65bx	65cx	65dx	65ex	65fx
65a0x		65a2x	65a3x	65a4x	65a5x	65a6x	65a7x	65a8x	65a9x	65aax	65abx	65acx	65adx	65aex	65afx

Figure 2: An example Routing Table [3]

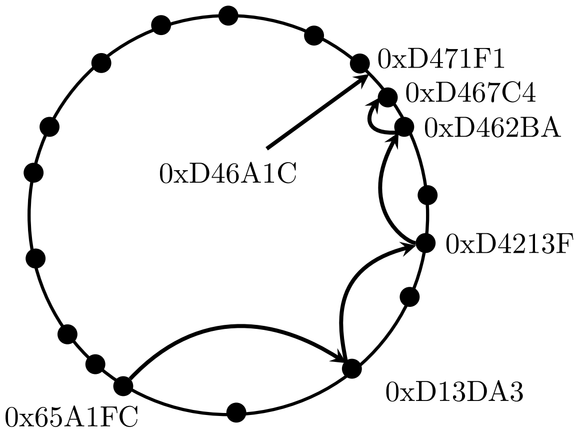


Figure 3: An example of routing a message from 0x65A1FC to 0xD64A1C

2.2 Multicasting

Scribe is application-level multicast infrastructure built on top of the fully decentralized peer to peer system, Pastry. It is scalable, self-organizing, and efficient in group communication [4]. Scribe builds a multicast tree for each group for efficient dissemination of messages. In this analysis, firstly, Pastrys short routes property and route convergence property is described briefly. After that, the architecture of Scribe is explained, and lastly, the test results about Scribes performance is shown.

Pastry has some locality properties such as the short routes property and the route convergence property. This effects Scribes performance especially in delaying messages and link stress in physical-level network. The short routes property is about the total distance (such as round trip time) that messages travel in the Pastry network, and the route convergence property is about the distance that two messages travel to the same key before their route converge. The average distance traveled by two messages before they converge is equal to the distance between their source nodes.

Scribe is a scalable and decentralized multicast infrastructure built on top of Pastry. It creates a group with a number of nodes that want to join the group. A Scribe node can multicast messages to all members of a group, or it can join a group. Scribe can support a large number of groups concurrently, and a Scribe node can be a member of multiple groups. Scribe provides best-effort delivery for sending messages, but it does not ensure that messages are delivered in order. However, the ordered delivery can be built on top of Scribe. To manage the members and groups, Scribe builds a multicast tree for each group. Like Pastry, Scribe is also fully decentralized. Any node can be a root, a group member, a multicast source, etc. or any combination of those.

Scribe offers an API. Some major methods are introduced. create(credentials, groupId) creates a group with credentials and groupId. The credentials are information to authenticate the local nodes and to securely join the Pastry network. join(credentials, groupId, messageHandler) makes the local node to join the group with groupId. The messageHandler is to handle subsequently received multicast messages for that group on the node. leave(credentials, groupId) makes the local node to leave the group that has the groupId. multicast(credentials, groupId, message) is to multicast the message within the group with the groupId. In addition, Scribe software on each node provides the forward and deliver method that are invoked by Pastry when the nodes receive messages. The forward method is called when a node needs to route messages through the node. The deliver method is called when a node receives messages with the nodeId that is numerically closest to the nodeId of the node that messages are arrived, or when messages are sent by Pastrys send operation to the local node.

A Scribe group has a unique groupId, and the node that has numerically closest nodeId to the groupId acts as the root of the multicast tree created for the group. The groupId is generated by the hash of the groups textual name concatenated with the nodes textual name that has created the group. A collision resistant hash function such as SHA-1 is used to compute the hash. This can make groupIds even distribution. A multicast tree is built for each group for dissemination of multicast messages. Multicast messages are disseminated from the rendez-vous point which is the root. Forwarders are the Scribe nodes in a multicast tree created for a group. The forwarders may or may not be members of the group. Each forwarder has a children table that contains an entry (IP address and nodeId) for each of its child nodes for the group. Forwarders are used when a node joins a group. They forward a join message to the root.

```
(1) forward(msg, key, nextId)
(2) switch msg.type is
(3) JOIN : if !(msg.group is an element of groups)
(4) groups = groups U msg.group
(5) route(msg, msg.group)
(6) groups[msg.group].children U msg.source
(7) nextId = null       // Stop routing the original message
```

Figure 4: Scribe implementation of forward.

When Scribe invokes the forward method, it checks if the node calling the forward method is currently a forwarder by looking at its list of groups (line 3 in Figure 4). If not, it creates an entry for the group, and adds the node that wants to join the group as a child in the children table (line 4, 6 in Figure 4). Also, it becomes a forwarder by routing a JOIN message to the the root (line 5 in Figure 4). If the node invoking the forward method is already a forwarder, it adds the joining node to the children table (line 6 in Figure 3). Then, the original message is terminated by setting nextId to null (line 7 in Figure 4). When a Scribe node leaves a group, it deletes its record in the children table. If there is no entry in the children table, it sends a LEAVE message to its parent nodes recursively until a node has entries in the children table.

The forwarding mechanism is similar to the Pastry routing mechanism. The nodeId of next node to forward the join message will have a longer prefix match or numerically closer to the groupId. This property ensures that there are no loops and a tree is produced. Also, the properties of randomization of Pastry balance the tree so that the forwarding load can be evenly distributed among the nodes. This results in large-scale management of groups in Scribe. The multicast tree is efficient to disseminate messages because of Pastry short route property. The delay in forwarding multicast messages from the root to a group is small because Pastrys routing algorithm minimizes the number of steps to forward messages. In addition, the route convergence property causes the small load on the physical network because most messages sent by the nodes that are close to their leaf nodes. Therefore, the network distance traveled by the messages is short.

For fault resilience in Scribe, non-leaf nodes in the tree send a message to their children periodically. If children fail to receive the message, it is assumed that their parent nodes are faulty. A node then sends a join message. Pastry will route the message to a new parent node to repair the tree structure. For the case that a root fails, the root is replicated across a number of its closest nodes (usually 5). These nodes are the members of the leaf set of the root node. When a root fails, its immediate children nodes detect that and send a JOIN message to a new root whose nodeId is numerically closest to the groupId. This node will take over the failed roots role. This tree repair mechanism is scalable. The number of node involved in fault resilience is (O(log 2b N)). By sending messages to a small number of nodes, failed nodes can be detected, and the recovery is local.

In experiments with 100,000 end nodes, Scribe scales well in a large number of nodes, groups, and a wide range of group sizes. It also balances the load among nodes in the network. In the delay penalty experiment comparing the RMD (the ratio of maximum delay) and RAD (the ratio of average delay) between Scribe multicast and IP multicast, 50% of groups RAD is 1.68 and RMD is 1.69 at most relative to IP multicast. It shows the short route property of Pastry. In the node stress experiment, the number of children tables (non-empty) and the total number of children table entries is reduced as the number of nodes is induced. With 1,500 groups, the mean number of children tables per node is 2.4 and the median is 2. The mean number of children table entries is 6.2 and the median is 3. This indicates that Scribe makes good distribution of loads.

In the link stress experiment, the mean number of messages sent to each link in Scribe is 2.4 and 0.7 in IP multicast. The median is 0 for each of them. In addition, the maximum link stress in Scribe is about 4 times greater than the one in IP multicast with the value of 4031 and 950 for each. This indicates that Scribe distributes load across nodes with Pastrys route convergence property.

Scribe can manage a large number of groups with a large number of members in each group simultaneously. The events of joining, leaving, and fault detection are locally processed. Also, the loads among groups or nodes in a tree or group are evenly distributed with Pastrys properties such as locality, short routes property, and route convergence property. For our project, fractal image generation on Pastry, we have developed software using Scribe because generating a fractal image can be done in a group efficiently. The nodes participating in fractal image generation can be the members of the group, and they can be managed by the Scribe infrastructure. In addition, our program is designed to multicast the final chunks of the image to every other nodes in the group. Scribe can give a suitable environment for our project.

2.3 Load Balancing

One important aspect of distributing a massively parallel computation is the problem of load balancing. The decision of which pieces of work should be assigned to which compute nodes usually has a significant impact on the total processing time needed to complete the computation. A load balancing strategy may also be able to improve the reliability of the computation by shifting the load to nodes which are more reliable. An ideal load balancing strategy would maximize throughput of data, while maintaining a high level of reliability. The design goals of our project present some additional constraints on the load balancing strategy.

First our cluster is comprised of heterogeneous nodes. Heterogeneous , or non-homogenous, nodes are nodes which do not share the same characteristics. Many parallel processing systems consist of homogeneous nodes which all have the same processing power, memory capacity, communication speed, and other characteristics. For example, in multiprocessor and multicore systems the nodes (cores) are typically all of the same characteristics (architecture, speed, connection speed). In cluster computing environments the nodes may also be homogenous. Our design is targeted toward systems that may be heterogeneous and require a strategy that can cope with nodes of different capability.

The design of our project also allows compute nodes to freely join or drop out of the group. The load balancing strategy must be able to recognize when a node is no longer available, stop assigning work to that node, and also make sure that any work assigned to that node which was not completed gets assigned to another node. On the other hand, the strategy must also handle new nodes joining the system at any time, and be able to assign them work appropriate to their capabilities.

Several research papers were examined which described different methods of load balancing [5, 6, 7, 2, 8]. The authors of [7] describe a system which compensates for delay in network communication, and an extension which scales back task execution when the node is processing other loads. In [2] a number of different load balancing strategies are reviewed and an asymmetric load balancing strategy is implemented and tested. These works each utilize a form of benchmarking to measure the performance capabilities of the nodes.

The primary load balancing work researched was [5]. This work describes three different strategies to balance load in heterogeneous clusters. They examine the N-Queens problem which requires a large number of states to be evaluated. To evaluate the performance of the strategies, they were implemented and run on a system of 25 heterogeneous nodes. A master-slave organization is used in which the master node distributes work to the slave nodes. The first, and simplest strategy described is called *Direct Static Distribution.* In this strategy the total workload is divided up into equal pieces and each node is given an equal share of the workload. The second strategy, *Predictive Static Distribution* uses a predictive function to evaluate the processing power of the node. The master node then distributes a chunk of work proportional to the amount of processing power to each node. The third strategy *Dynamic Distribution upon Demand* distributes work to the nodes on demand. The predictive function is used to measure the performance of each node, then an initial, smaller proportion of the total work is distributed to each node, as in the *Predictive Static Distribution.* However, once each node is finished with its work it requests a new chunk of work from the master. The master distributes a fixed amount of work to each requesting node, and decreases this fixed amount each time. Figure 5 shows an example of a distribution using *Dynamic Distribution upon Demand.* The authors used a selection of metrics to measure the performance of their implementations. The *Unbalance* metric, shown in Equation 1, measures the difference between the longest amount of time spent processing and the shortest. In order to achieve an optimal speed-up, it is desirable for each machine to spend the same amount of time processing, and thus no machine will be idle while another is processing. Figure 6 shows the load unbalance measurements for each of the load balancing strategies. The multiple columns under Dynamic on Demand represent a selection of parameter values. The first value is the percentage of the total work that is distributed in the initial step. The second value is the fixed amount that is distributed to each node on demand. Figure 7 shows a comparison of the different load balancing strategies using the load unbalance metric. Here the Dynamic upon Demand strategy is implemented with 25% initial work distribution and a 1 unit distribution on demand. Clearly the *Dynamic Distribution upon Demand* strategy dominates the other strategies in all cases tested. The speed-up achieved using each of these strategies is compared in Figure 8. The authors neglect the impact of communication overhead on the performance of these strategies.

$$Unbalance = \frac{\max_{i=1,\ldots,B}(W_i) - \min_{i=1,\ldots,B}(W_i)}{\text{avg}_{i=1,\ldots,B}(W_i)} \tag{1}$$

In [8] a slightly different approach is taken. In this work, the master maintains a queue of nodes available to compute, and a list of nodes currently doing work. As the master distributes work it takes a node off of the queue and places it on the list of nodes currently doing work. When that node has completed the work it is taken off the list and placed on the end of the available nodes queue. This simple strategy has many advantages for our application.

In our work, we implement a load balancing strategy similar to the work of [8] and the *Dynamic Distributed on Demand* strategy of [5]. The main design goals concerning load balancing are performance and fault tolerance. The Dynamic on Demand strategy possesses many qualities which are desirable for our project, but a few which are problematic. A demand driven strategy allows a simple way to avoid many of the complexities of maintaining a node state and recovering from failures. If a node fails after it has joined the computation, it will no longer request more work, and therefor no more work will be assigned to it. Care must be taken to make sure any work a failed node was processing or has completed is not lost. When a new node joins after the computation has started, it can simply demand and receive work from the master without adjusting the workload of the other nodes. In order to simplify the implementation of the strategy we excluded the use of a predictive performance function. The complete design and implementation is detailed in Section 3

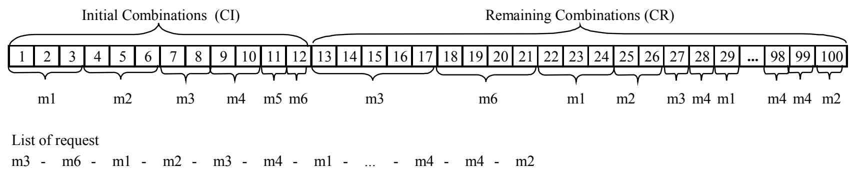

Initial Combinations (CI)
Remaining Combinations (CR)
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 ... 98 99 100
m1 m2 m3 m4 m5 m6 m3 m6 m1 m2 m3 m4 m1 m4 m4 m2
List of request
m3 - m6 - m1 - m2 - m3 - m4 - m1 - ... - m4 - m4 - m2

Figure 5: *Dynamic Distribution upon Demand* example [5]

Size	Direct Static	Predictive Static	Dynamic upon Demand					
			0% - 1	0% - 5	0% -10	25% - 1	25% - 5	25% -10
17	93%	45%	11%	11%	10%	11%	9%	61%
18	98%	23%	9%	10%	11%	11%	11%	11%
19	130%	58%	8%	9%	9%	9%	9%	9%
20	88%	32%	6%	6%	6%	8%	7%	37%
21	110%	29%	7%	6%	6%	5%	6%	4%

Figure 6: Table of *Unbalance* measurements [5]

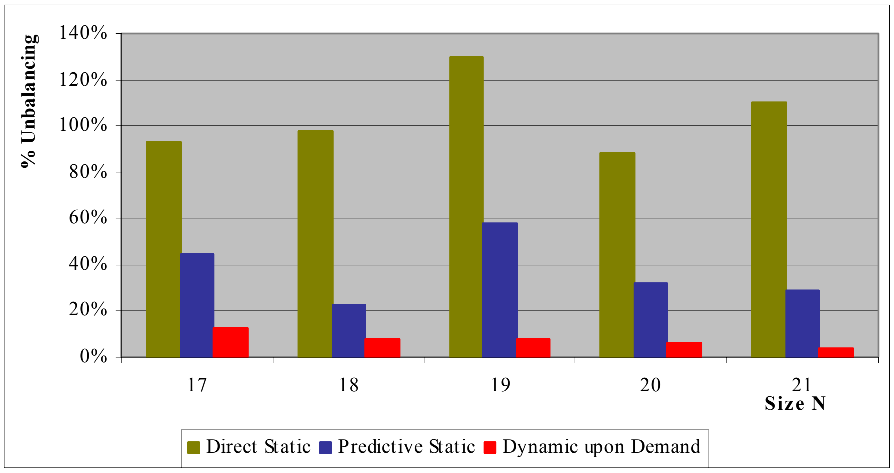


Figure 7: Comparison of load balancing strategies using *Unbalance* metric [5]

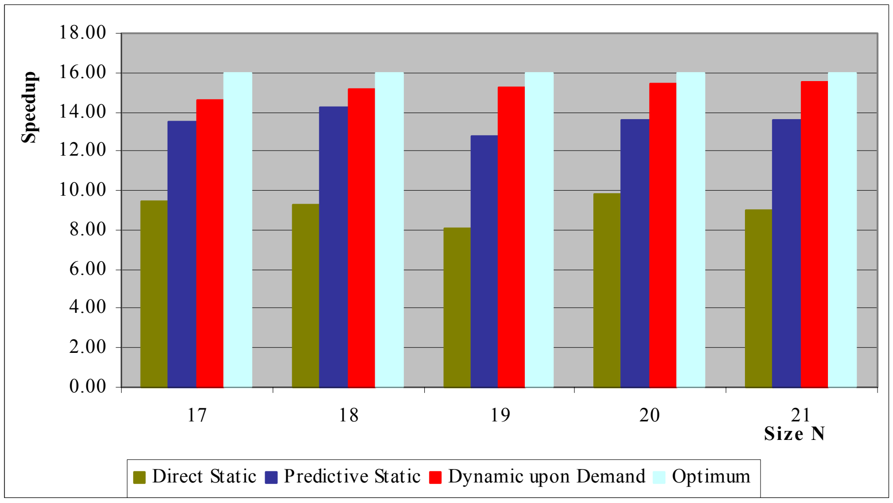


Figure 8: Comparison of load balancing strategies using *Speed-up* metric [5]

3 Design

This section describes the details of the software and algorithm design. Section 3.1 describes the implementation of the fractal generation library. Section 3.2 describes how the fractal generation application is parallelized using a client-server model. Section 3.3 describes how the parallelized task is distributed across a peer-to-peer network.

3.1 Fractals

Fractals are geometric entities which possess a quality called *self-similarity*. This means that certain regions of the figure are similar to the entity as a whole. This application is concerned with fractal images which can be generated by calculating the value of each pixel independently. In particular, a special case of fractals called *escape-time fractals* or *orbit-trap fractals* are examined. In these fractals the value of each pixel is based on a recurrence-relation, which is solved using a recursive or iterative algorithm. Sections 3.1.1 and 3.1.2 describe two fractals which were implemented for this application.

3.1.1 Mandelbrot Set

The Mandelbrot Set was discovered by Benoît Mandelbrot. The set is defined as the set of complex numbers whose orbit remains bounded under iteration of the function given by Equation 2, where c is a complex number parameter. When complex number are part of this set, the value of this function may “orbit”, but it remains bounded under any number of iterations. This fractal can be drawn by converting the complex plane into the Cartesian plane, where the x-axis represents the real part of the complex number, and the y-axis represents the imaginary part of the complex number. Each coordinate on the plane then represents a complex number c which is used as the parameter to the function in Equation 2. The initial value of c is the coordinate of the pixel being generated. Aesthetically, the interesting part of the fractal image lies on the border between the points which are in the set and those which are not. These points do not have a bounded orbit, but exit the bound at different times during the iteration. By associating a color with the number of iterations required for the orbit to break a given bound the fractal can be colored. Figure 9 shows an example of a Mandelbrot fractal generated with this fractal.

$$f_c(z)\colon z \mapsto z^2 + c \tag{2}$$

More advanced methods of coloring are discussed in [1]. For the implementation of the Mandelbrot fractal generation class, a continuous coloring algorithm based was used as shown in Equation **??**. This continuous value was then used to linearly interpolate a color value from a discrete indexed palette. In this implementation the points inside the set are also colored with this scheme.

$$C(z)\colon z \mapsto z + e^{|z|} \tag{3}$$

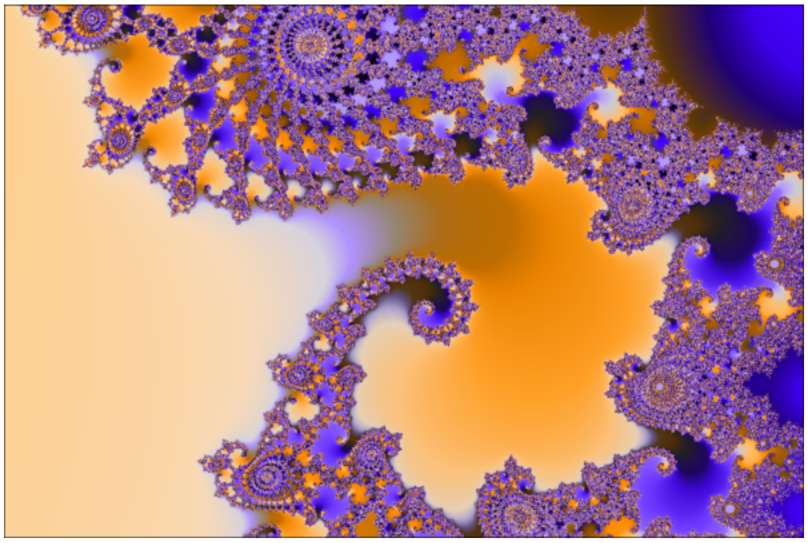


Figure 9: Mandelbrot Fractal

3.1.2 Julia Set

The Julia Set is strongly related to the Mandelbrot Set discussed in the previous section, and is calculated in much the same way. The same equation, Equation 2 is used, however the complex number parameter c is fixed for the entire image, as opposed to being the coordinate of the pixel as in the Mandelbrot Set. Thus different fractals are generated for different parameters c. The initial value of z is the coordinate of the pixel being generated. The Julia fractal generator implemented for this work uses a discrete valued coloring scheme based on a palette. Figure 10 shows an example of a Julia fractal generated with this project.

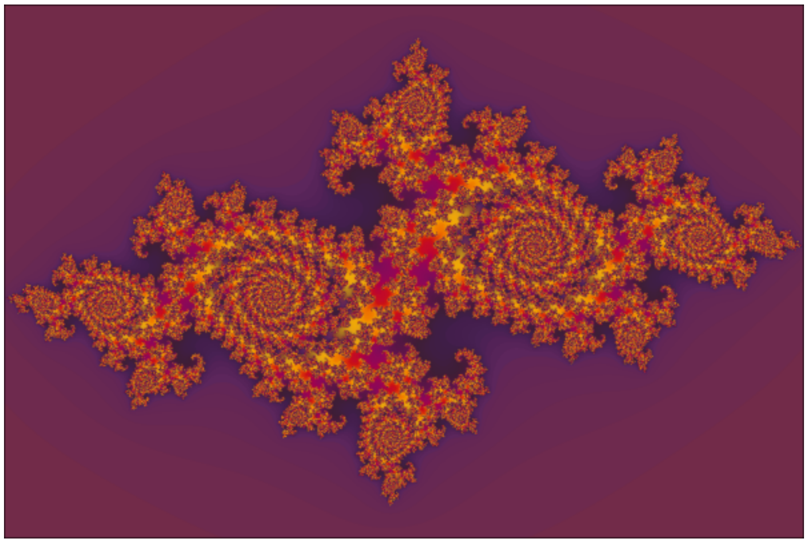


Figure 10: Julia Fractal

3.2 Parallel Processing

The first step in distributing the rendering of fractal images is to parallelize the task. To accomplish this, a simple client/server model was developed. A single server is run which acts as the master node. Client nodes, or slaves, connect to the master node through sockets to communicate. The master node begins by dividing up the image into approximately equally sized ranges of pixels. A parameter controls the granularity, the number of ranges. Finer grained control leads to less load imbalance, but increases the proportion of overhead. These ranges of pixels paired with the parameters to the particular instance of the fractal, make up a slice of work. The master node keeps a queue of the work which needs to be done and distributed this work to the client nodes on demand. The client nodes connect to the server node, and immediately the server sends a work assignment to the client. The client node then sends data back to the master node and the master node compiles this data into an image. If there is work remaining in the queue, another slice is distributed to the client immediately. In this implementation only the master node gathers the final image, although it could easily distribute this to the clients at a later time.

There are several problems with this implementation which are addressed with the distributed implementation. It suffers from the single point of failure that a client/server model exhibits. It is not fault tolerant, if a worker node dies during the computation, any work it was currently working on is lost. If the master node fails, all nodes fail and the computation fails. Also, the master node must maintain an open socket to each of the worker nodes. As the number of worker nodes grows, the number of open sockets and threads the master must keep also grows, linearly. The worker nodes must be aware of a single server address, and that server must have high availability and enough resources to handle many open communication channels.

3.3 Distributed Processing

In order to address the problems presented by the client/server parallel processing model described above, a distributed parallel processing model was developed. The main design goals of the system were to achieve reasonable performance while maintaining a high degree of fault tolerance. Worker nodes should be able to connect to the processing group easily, and be able to drop out of the group with minimal impact on the rest of the nodes. A decentralized architecture allows any of the worker nodes to assume the role of master, and allow role migration to occur dynamically when needed. Distribution of the communication overhead results in more efficient scaling as the group size increases, as well as reducing the amount of work any one node has to do to deal with nodes joining and dropping.

The design of the distributed system differs in a few key ways from the client/server model:

- There is no static master, the role of master may change during the course of processing.
- All clients connecting to the processing group will receive the slices of work as they are completed.
- When a node fails, the work is not lost.
- When a master fails, another node in the group may assume the role of master.
- As the number of nodes increases, the number of communication channels increases logarithmically.

To accomplish this the Pastry peer-to-peer overlay and Scribe multicast frameworks were utilized. The details of Pastry are discussed in Section 2.1 and Scribe is discussed in

Section 2.2. Nodes may first bootstrap using any of the nodes already in the processing group. If there are no nodes existing, then the node starts the group, and assumes the role of master. As other nodes join in the group they connect to a node in the processing group, and also subscribe to the Scribe tree of the group. Nodes broadcast a *MasterQuery* message until they receive a *MasterResponse* message from the master node. The *MasterResponse* message contains the nodeId of the master, as well as a copy of the work already completed, the work which still needs to be done, and other information necessary for the node to assume the master role, if needed. At this point the node communicates directly (using Pastry) with the master node and sends a *WorkRequest* message. When the master receives a *WorkRequest* message it checks to see if there is more work to be done. If there is more work it responds to the node with a *WorkAssignment* message which contains the parameters of the fractal and a range of pixels to calculate. This slice of work is removed from the head of the queue and placed on the back. When a node completes the processing on a range of pixels it broadcasts a *FractalData* message. Any other nodes in the same processing group receive this message and render this slice of the fractal. Since each node is also maintaining its own list of work which needs to be completed, the work slice is removed from this list. The worker node then requests more work with a *WorkRequest* message to the master. In this fashion the nodes continue to process work assignments from the master until all work is done. The node joining process is shown in Figure 11, and the work assignment process is shown in Figure 12.

It is possible in this scheme that work may be completed after the master sends out a *MasterResponse* message, but before a node receives it. At this point the new node would be out of sync with the other nodes in the group. This node will still have work in its queue when the other nodes of the group have finished processing. For this reason, when the master node is done with its work it relinquishes its role as master and an election occurs. Any node that still has work to do may participate in the election, and if it becomes master, distribute its work to complete the task.

There are two cases of failure that must be handled in order for the system to recover from a fault. In one case a worker node fails. It may fail while it is processing a slice of work, or it may fail during any state of message passing. In order to make the master resilient against node failure, when the master distributes work to a node, it puts that slice of work at the end of the queue. If a worker node drops and fails to return the data associated with this slice, the master node will assign it to a different node when it reaches the front of the queue again. This strategy has the added benefit that if there is little work to be done, multiple nodes may work on the same slice of work and when the faster node completes, it is distributed, and when the slower node completes, no harm is done. This happens only when there is a smaller number of slices in the queue than there are nodes in the ring, so no work is starved in this case. The other case of failure is the case when a master node fails. When this happens the other nodes in the group must elect a new master to distribute the work. Since each of the nodes has a loosely synchronized copy of the work that needs to be done, any of the node with more work to be done may be a useful master. When a node is notified by Pastry that another node has dropped, and recognizes that the dropped node was the master node, it initiates an election.

The election process is initiated when the current master node has no more work to complete, or the current master node drops from the group. Scribe helps the election process by organizing the nodes of the group into a tree. Since there is only one root of the tree, this node can decide who should be elected as the next master with no ambiguity. The election is initiated by a node broadcasting an *ElectionQuery* message. When this message is received by a node with more work to be done, that node broadcasts a *MasterNomination* message, nominating itself as a reasonable candidate. If a node does not have more work to do, it does not send any message. When the root of the Scribe tree receives an *ElectionQuery* message it is put into an *electing mode*. While in *electing mode* if the root node receives a *MasterNomination* message it immediately declares that node to be the master by broadcasting a *MasterElection* message and removes itself from the *electing mode*. Thus the master is chosen by the root of the Scribe tree on a first-come-first-serve basis. The election process is demonstrated in Figure 13.

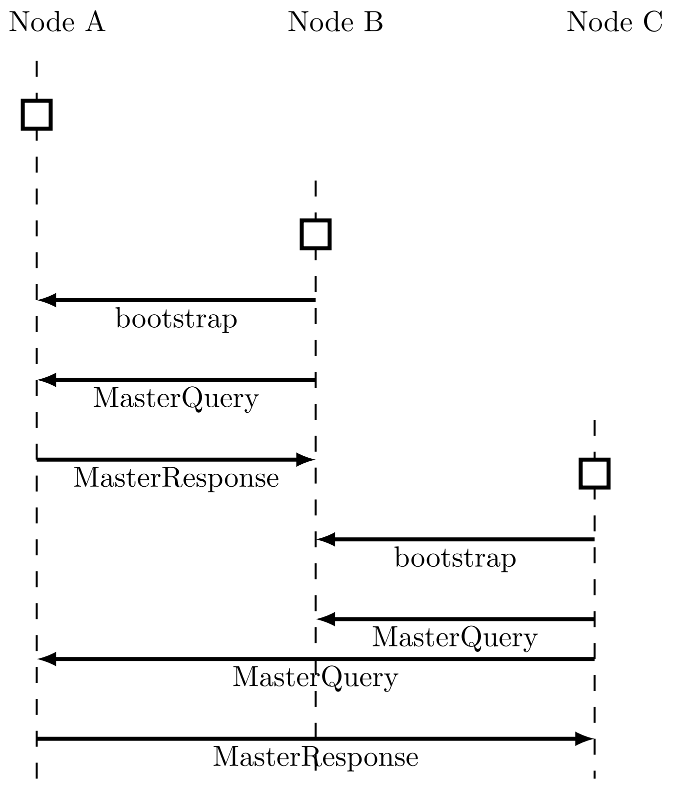


Figure 11: Node Join Process

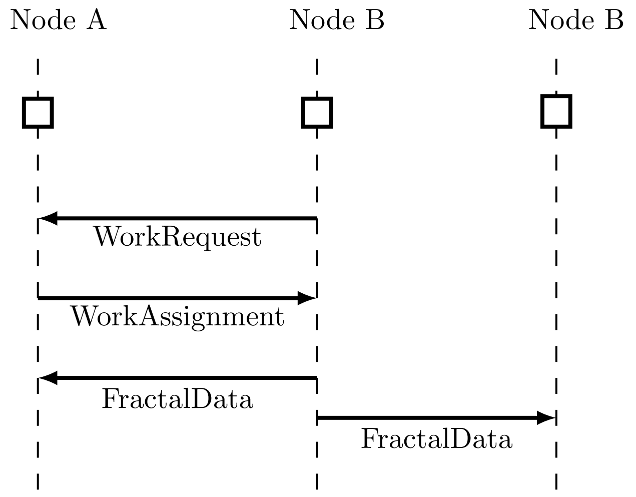


Figure 12: Work Assignment Process

4 Developer Manual

To compile the source code and run the project execute the following steps.

1. Extract the jar `mppp2p.jar` into a directory of your choice.
2. Execute the following command to compile the FractalGen Library:
   `javac FractalGen/src/*/*.java`
   Alternatively you may change directory into the package directories and compile the java source files directly.
3. Execute the following command to compile the MPPP2P Applications.
   `javac -cp FractalGen/src:FreePastry-2.0_03.jar MPPP2P/src/*/*.java`
   Alternatively you may change directory into the package directories and compile the java source files directly.

5 User Manual

In order to run the applications in the MPPP2P project the source must first be compiled, as described in the previous section. Once the source is compiled execute the following steps to run the application of your choice. All commands shown here are executed from the `MPPP2P` directory.

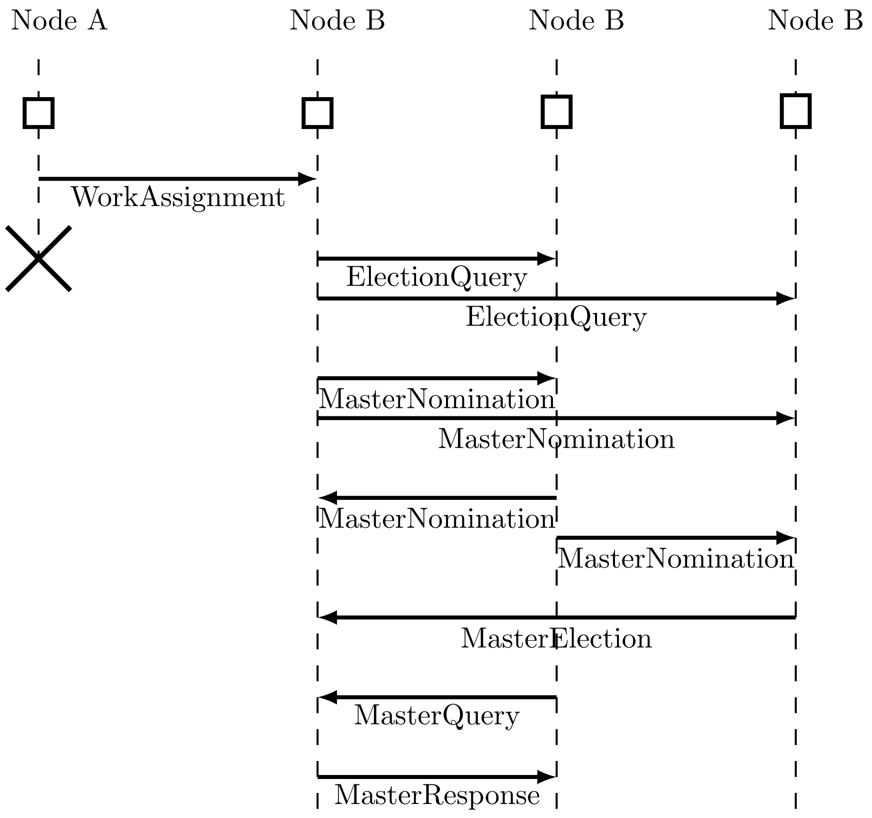


Figure 13: Election Process

5.1 Client/Server Model

First the master node must be executed by running the command:
`java -cp ../FractalGen/src:../FreePastry-2.0_03.jar:src clientserver.StaticMaster <port> <slices>`
where `<port>` is the number of the port you wish to listen on, and `<slices>` is the number of slices to divide the data into.

Starting the master node should produce output similar to the following:

```
java -cp ../FractalGen/src:../FreePastry-2.0\_03.jar:src
 clientserver.StaticMaster 9000 10
added: (0..1800, 0..120)
added: (0..1800, 120..240)
added: (0..1800, 240..360)
added: (0..1800, 360..480)
added: (0..1800, 480..600)
added: (0..1800, 600..720)
added: (0..1800, 720..840)
added: (0..1800, 840..960)
added: (0..1800, 960..1080)
added: (0..1800, 1080..1200)
```

To run each worker node execute the following command:
`java -cp ../FractalGen/src:../FreePastry-2.0_03.jar:src clientserver.FractalWorker <host> <port>`
where `<host>` and `<port>` identify the socket address of the master node run in the previous command.

Running a worker node should produce output similar to the following:

```
java -cp ../FractalGen/src/:../FreePastry-2.0_03.jar:src
 clientserver.FractalWorker localhost 9000
starting 0
client0: connecting to localhost:9000
client0: connected.
client0: opened.
client0: awaiting work...
client0: processing... (0..1800, 0..120)(java.ServiceProvider)
client0: sending data...
client0: 1971ms
client0: awaiting work...
client0: processing... (0..1800, 120..240)
```

```
client0: sending data...
client0: 1811ms
client0: awaiting work...
client0: processing... (0..1800, 240..360)
client0: sending data...
client0: 1944ms
client0: awaiting work...
client0: processing... (0..1800, 360..480)
client0: sending data...
client0: 3395ms
client0: awaiting work...
client0: processing... (0..1800, 480..600)
client0: sending data...
client0: 5553ms
client0: awaiting work...
client0: processing... (0..1800, 600..720)
client0: sending data...
client0: 6153ms
client0: awaiting work...
client0: processing... (0..1800, 720..840)
client0: sending data...
client0: 6400ms
client0: awaiting work...
client0: processing... (0..1800, 840..960)
client0: sending data...
client0: 4742ms
client0: awaiting work...
client0: processing... (0..1800, 960..1080)
client0: sending data...
client0: 4275ms
client0: awaiting work...
client0: processing... (0..1800, 1080..1200)
client0: sending data...
client0: 3847ms
client0: awaiting work...
```

and the master node terminal should produce output similar to the following:

```
received: (0..1800, 0..120)
received: (0..1800, 120..240)
received: (0..1800, 240..360)
received: (0..1800, 360..480)
```

```
received: (0..1800, 480..600)
received: (0..1800, 600..720)
received: (0..1800, 720..840)
received: (0..1800, 840..960)
received: (0..1800, 960..1080)
received: (0..1800, 1080..1200)
```

When the work is complete, the master process will write an image with the filename `out.png`, and both processes will terminate. Multiple nodes can be started in the same fashion on the same machine or different machines and participate in the same processing group by specifying the socket address of the master node start in the first step.

5.2 Peer-to-Peer Model

Nodes are run by executing the following command:
`java -cp ../FractalGen/src:../FreePastry-2.0_03.jar:src p2p.FractalWorker <localport> <host> <remoteport>`
where `<localport>` is the port that this node will listen on for other nodes, `<host>` and `<remoteport>` identify the socket address of the node already in the ring to bootstrap to. If this is the first node in the ring, you can simply specify `localhost` and `0` for the host and port.

Running the first node should produce output similar to the following:

```
java -cp ../FractalGen/src/:../FreePastry.0_03.jar:src/ p2p.FractalWorker
9000 localhost 0
:rice.pastry.socket:1203823755803:Error connecting to address
 localhost/127.0.0.1:0: java.net.BindException: Can't assign requested address
:rice.pastry.socket:1203823755815:No bootstrap node provided,
 starting a new ring binding to address flashtop.rit.edu/129.21.40.23:9000...
registered (<0x2D2C33..>)
added: (0..1500, 0..50)
added: (0..1500, 50..100)
added: (0..1500, 100..150)
added: (0..1500, 150..200)
added: (0..1500, 200..250)
added: (0..1500, 250..300)
added: (0..1500, 300..350)
added: (0..1500, 350..400)
added: (0..1500, 400..450)
added: (0..1500, 450..500)
added: (0..1500, 500..550)
```

```
added: (0..1500, 550..600)
added: (0..1500, 600..650)
added: (0..1500, 650..700)
added: (0..1500, 700..750)
added: (0..1500, 750..800)
added: (0..1500, 800..850)
added: (0..1500, 850..900)
added: (0..1500, 900..950)
added: (0..1500, 950..1000)
ok...
```

To run subsequent nodes, specify the port used as the remoteport in the last command and the host that node is running on. The output should look something like the following:

```
java -cp ../FractalGen/src/:../FreePastry.0_03.jar:src p2p.FractalWorker
 9001 129.21.40.23 9000
registered (<0x79F066..>)
ok...
class p2p.MasterResponse
* acquired master <0x549A75..>
class p2p.MasterQuery
requesting work...
waiting for work...
class p2p.WorkAssignmentMessage
processing work (0..1500, 0..50)...
sending work...
class p2p.FractalDataMessage
gathered (0..1500, 0..50) from <0x79F066..>
requesting work...
waiting for work...
class p2p.WorkAssignmentMessage
processing work (0..1500, 50..100)...
sending work...
class p2p.FractalDataMessage
gathered (0..1500, 50..100) from <0x79F066..>
requesting work...
waiting for work...
class p2p.WorkAssignmentMessage
processing work (0..1500, 100..150)...
sending work...
class p2p.FractalDataMessage
gathered (0..1500, 100..150) from <0x79F066..>
```

```
requesting work...
waiting for work...
class p2p.WorkAssignmentMessage
processing work (0..1500, 150..200)...
sending work...
class p2p.FractalDataMessage
gathered (0..1500, 150..200) from <0x79F066..>
requesting work...
waiting for work...
```

and the first node run should produce the following output:

```
node ([SNH: <0x79F066..>//129.21.40.23:9001 [5389810429716979075]]) joined
class p2p.MasterQuery
class p2p.WorkRequestMessage
distributing work (0..1500, 0..50) to <0x79F066..>
class p2p.FractalDataMessage
gathered (0..1500, 0..50) from <0x79F066..>
class p2p.WorkRequestMessage
distributing work (0..1500, 50..100) to <0x79F066..>
class p2p.FractalDataMessage
gathered (0..1500, 50..100) from <0x79F066..>
class p2p.WorkRequestMessage
distributing work (0..1500, 100..150) to <0x79F066..>
class p2p.FractalDataMessage
gathered (0..1500, 100..150) from <0x79F066..>
class p2p.WorkRequestMessage
distributing work (0..1500, 150..200) to <0x79F066..>
```

Each node will write an image named after their *nodeId* when they have acquired all slices of the image. The nodes will continue to request work in case there are any other nodes in the group with more work to, or a new node joins with work to do (not implemented in this release). Typing `q<return>` will cause the node to end.

6 Results

This section discusses the functional analysis of our system in terms of job completion time and fault tolerance. Since the network model of our system is built upon Pastry, the system is by nature has a efficient routing performance. In the earlier section of the paper where we reviewed the Pastry architecture we saw Pastry performing quite gracefully in containing the number oh hops for a message between the source and destination to be $\lceil \log_{2^b} N \rceil$ in the environment where the number of nodes is varied from 1000 to 100,000.

One another such experimental result shows that the probability of number of hops remains to be very close to the predicted value of $\lceil \log_{2^b} N \rceil$ in a setup that has 100,000 nodes and 200,000 random lookups have been carried out. Figure 16 shows the logarithmic growth of the number of hops. Further analysis of the experimental results shoe that Pastry shows optimal performance in case of maintaining the network when randomly a large amount of nodes are made to fail simultaneously.

Performance analysis of our system is one of the most important section of our future developments. At the current state our system is not feasible for any time constrained results because as we mentioned earlier there has been a considerable amount of latency in the delivery of communication messages. Also other than taking a considerable time, these messages have a random time of arrival. This makes any type of time constraint comparison non conclusive. Once we have established a reliable communication mechanism between our nodes, we would come up with a detailed performance analysis in which we would put a system to rigorous test measuring its scalability, fault tolerance and routing efficiency.

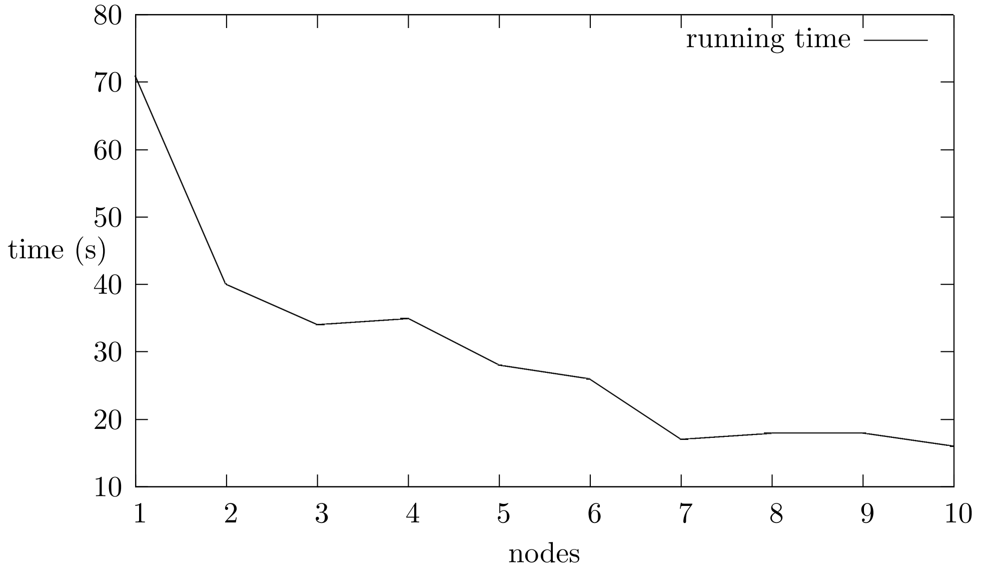


Figure 14: Running Time vs. Number of Nodes

7 Future Work

In our work on Massively Parallel Computing on Peer-to-Peer networks we came across several aspects of the peer-to-peer architectures that were quite flexible as well as complex. With the help of Free Pastry API from Rice University, we were able to create a system that

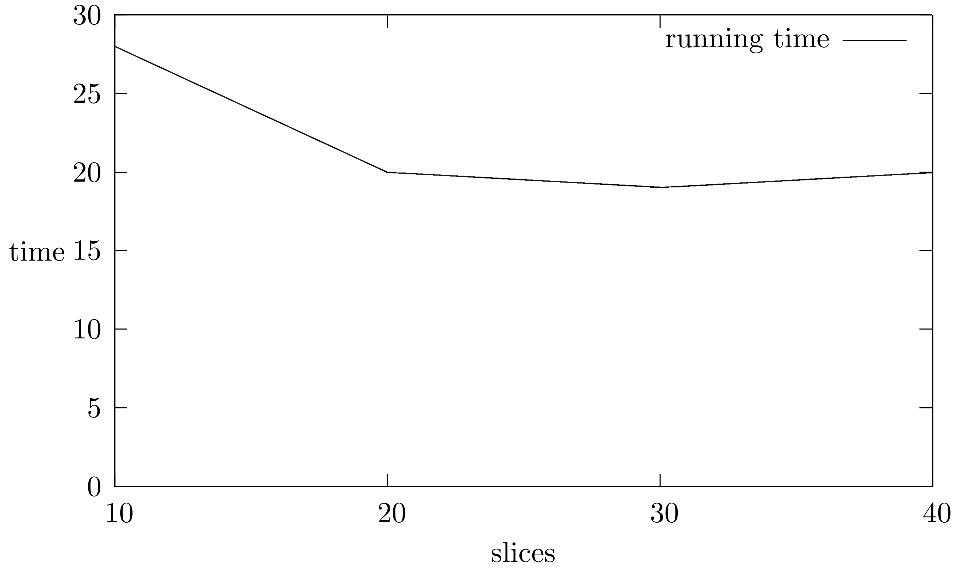


Figure 15: Running Time vs. Slices of data (with 5 nodes)

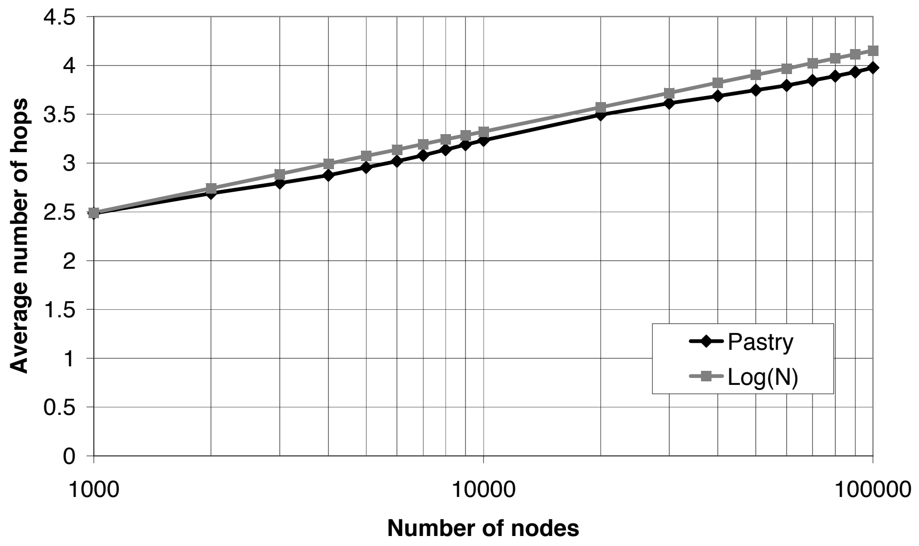


Figure 16: Logarithmic Growth [9]

effectively implemented work distribution, was highly scalable, had fault tolerance qualities and was totally decentralized. However there were certain areas that we couldnt work upon due to complexity and time restriction. This section describes few such aspects that we intend to work upon in future to make our application more adaptive and acceptable.

The architecture of our system was so designed that it restricted the work distribution to a single master and multiple worker nodes in the system. Though this approach was acceptable for a smaller ring with approximately 10 nodes, for a larger system with 1000s of nodes this approach could be an overkill. Depending upon the amount of resources required to complete the job, there may arise a situation that splitting one single work amongst 1000s of worker nodes would end up adding extra overhead and would decrease the efficiency of the system. Our design was biased to elect the first node creating the ring as the master node and all further subsequent nodes added as the worker nodes. The idea to such role assignment was partially borrowed from Scribe architecture, where nodes can subscribe to different Topics. Later the master node of that sub group in the ring could communicate with its own set of worker nodes through multicast or anycast. To keep the complexity to a moderate level, our approach registered all the nodes to a default Topic, thereby all nodes working together as a large group. The next step towards improvement in this area could be nodes registering for multiple topics thereby having multiple master-worker node sets in the ring. Since our system is already implemented on the steps of Topic subscription and each node is independent of other nodes, with all nodes having the same capability, this addition would require minimal changes to the architecture. Once this change has been made, our application would be capable of handling a wide granularity of jobs.

Another limitation in our work comes inherent from Pastry architecture, a substantial latency in delivery of messages, specially in case of node failure alerts. The implementation of Pastry limits the isAlive() function that checks for the presence of a neighboring node to fire after a considerable amount of time. This restriction forced our application to know about a nodes departure after 20 seconds, without regards of the number of nodes in the ring. This latency though may be acceptable for some applications, it increased our job completion time to a significant amount. In the future version of our work we intent to minimize this latency in message delivery. How we plan to go about it is by the use of application level Socket interface. The node that intends to establish one such interface would first route a message to the destination node with its own handle in the message section. The destination node than in return would request a socket connection to the previous node. Once the required authentication gets completed, these nodes than can communicate directly through socket read and writes. This would save the time that a message takes in the unnecessary hops that it takes while traveling to the destination.

The next minor modification that we would like to make to our project is in terms of a verbose notification to the end user in the role migration process. As per our implementation of the role migration module, the scribe tree is used to elect a new master amongst the existing worker nodes. Sometimes due to a major number of node failures, some message gets lost. As this implementation works mostly under hood of the Scribe implementation, the end user remains clueless of what is going in the background. Our modification would require a detailed study of Scribe architecture so that some internal modules could be added to it so as to inform the user as to what is going in the background. this would further increase end user acceptability of our application.

At last we would like to have a detailed performance analysis of our system compared to system that have a client-server architecture or peer-to-peer systems that do not implement work distribution. Our application of fractal generation required a huge amount of computational resources. This application when run on a system that does not implement work distribution can tend to overwhelm the system. We have performed some initial form of comparison.

8 Lessons

Our project encompassed some of the very interesting aspects of peer-to-peer architecture. Unlike client/server architecture, peer-to-peer systems have a wide range of capabilities that can be explored to attain a high level of flexibility in ones application. However with this great potential comes along some very complex algorithms and concept that broadens ones knowledge domain. During our project we came across several P2P architectures, we also analyzed some of these architectures like Pastry and Scribe in depth. We came across several algorithms, like Centralized Directory Model, Flooded Request Model and Document Routing Model on which these systems are based. We incorporated the Document Routing Model for our project in form of Pastry. Also the decentralized, scalable, anonymous, self-organizing and fault resilient properties of the peer-to-peer systems gave us lots of scope to explore some new concepts.

In our analysis of Pastry and scribe architecture, we came across some very well designed architectures. These designs and implementation of such complex systems was a good exposure to design and work with such systems. Our project used the Pastry API provided by Rice University called Free Pastry. Analysis of this API and then extending it suite our projects need was a great learning experience. While designing our architecture we also reviewed several load balancing and work distribution strategies to implement out master-worker model. To confine the complexity of our architecture we choose to implement a Static Load Balancing algorithm in our project. This lead to assignment of work to the nodes before the start of actual processing. For our master-worker model we also came across several work distribution algorithms. after a careful amount of consideration we implemented a direct distribution algorithm where all the node are considered to be equally equipped and assigned the same amount of task. Overall working on this project was a great boost on our learning curve and gave us a good in-depth knowledge of distributed systems.

References

[1] J. Barrallo and DM Jones. Coloring Algorithms for Dynamical Systems in the Complex Plane. *Visual Mathematics*, 1.

[2] C. A Bohn and G. B. Lamont. Load Balancing for Heterogenous Clusters of PCs. *Future Generation Computer Systems*, 18(3):389–400, 2002.

[3] M. Castro, P. Druschel, Y.C. Hu, and A. Rowstron. Topology-aware routing in structured peer-to-peer overlay networks. *Lecture notes in computer science*, pages 103–107.

[4] M. Castro, P. Druschel, A.M. Kermarrec, and AIT Rowstron. Scribe: a large-scale and decentralized application-level multicast infrastructure. *Selected Areas in Communications, IEEE Journal on*, 20(8):1489–1499, 2002.

[5] A. E. De Giusti, M. R. Naiouf, L. C. De Guisti, and F. Chichizola. Dynamic Load Balancing in Parallel Processing on Non-Homogeneous Clusters. *JCS&T*, 5(4), December 2005.

[6] A.C. Dusseau, R.H. Arpaci, and D.E. Culler. Effective distributed scheduling of parallel workloads. *ACM SIGMETRICS Performance Evaluation Review*, 24(1):25–36, 1996.

[7] C.C. Hui and ST Chanson. Improved strategies for dynamic load balancing. *Concurrency, IEEE [see also IEEE Parallel & Distributed Technology]*, 7(3):58–67, 1999.

[8] R. Labriaga and D. Williams. A Load Balancing Technique for Heterogeneous Distributed Networks. *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, 2000.

[9] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, 11:329–350, 2001.