

RAIDS

Redundant Array of Independent Distributed Storage Systems

Team:
Distributed Monotonicity
Members:
Kevin Cheek, Alex Maskovyak, Joe Pecoraro

Project Description

RAIDS, or Redundant Array of Independent Distributed Storage Systems, is a per-file distributed backup system. It offers RAID-5-like (Redundant Array of Independent Disk) error detection and recovery on a distributed system of computers. RAIDS stores files in pieces across multiple systems. The use of striped parity bits allows for file recovery in the event that all copies of a piece are irretrievably lost. RAIDS makes use of FreePastry, an open-source peer-to-peer system framework. FreePastry handles file-tracking, reliable redundancy across the "cloud", efficient routing, and network maintenance chores. RAIDS implements FreePastry's Application interface, allowing it to be installed on any logical node and to make use of the FreePastry network substrate. RAIDS uses PAST, a distributed hash table application built for FreePastry, as its primary means of storing file backup meta-information. RAIDS also uses Scribe, a decentralized subscription/publishing application built for FreePastry for multicasting storage requests.

Analysis of Paper 1

A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems". IFIP/ACM International Conference on Distributed Systems Platforms (Middleware), Heidelberg, Germany, pages 329-350, November, 2001. <http://research.microsoft.com/en-us/um/people/antr/PAST/pastry.pdf>

This paper gives a full description of the Pastry peer-to-peer framework. It covers many of the implementation details concerning Pastry's reliable and efficient overlay network. We chose this paper so that we could obtain a solid understanding of how the underlying peer-to-peer substrate works in FreePastry. This information allowed us to properly leverage the system for our own application.

The paper begins by explaining the purpose behind Pastry's construction. At the time the paper was written and Pastry was developed (2001), existing peer-to-peer implementations still required some form of centralization. The authors cite applications such as Napster, Gnutella and FreeNet; claiming that they had peer-to-peer networks that were not completely decentralized. The goal of Pastry was to provide a completely decentralized peer-to-peer network. They described this as a system in which "all nodes have identical capabilities and responsibilities and all communication is symmetric." Furthermore, they intended Pastry to solely be a "general substrate for the construction of a variety of [...] applications"; on top of which any desired peer-to-peer application could be built. A few of these applications, PAST and SCRIBE, are mentioned throughout the paper. We studied these two applications in particular and were able to exploit their functionality.

Pastry is self-organizing and fault-tolerant. Its efficient routing algorithms take locality into account which makes it highly scalable. Pastry routes messages in $\log(N)$ steps

(hops on nodes in the network). What follows is an explanation of node identification, a description of the node routing table, and routing protocol itself.

The nodeIds in the paper are described as being 128 bits, thus allowing ids to range 0 to $2^{128}-1$. Upon joining the system, a node is assigned a random id. By way of a hash function the ids can be assumed to be uniformly distributed across the 128 bit ring-space. This hashing means that no significant data can be derived from the node id such as locality or proximity. Nodes with very similar ids may be on entirely different sides of the world or system. This also means that the range can be assumed to be split up rather uniformly covering the entire range.

Routable messages have a key, or searchId, that falls in this same 128 bit range. The objective is to find the node in the peer-to-peer network that has the nodeId that is numerically closest to that searchId. The core principle in Pastry's routing scheme is to ensure that a message moves "closer" to the destination node on each hop. Each node forwards the message to the next node that it knows is closer to searchId. This continual improvements is very effective. It is achieved by taking advantage of the format of the nodeIds. As a message gets closer, more and more of the searchId's MSBs (most significant bits) will match the destination nodeId's MSBs. These bits are referred to as the "prefix" and may make more sense if you represent the bits in their hex string representation. Just how Pastry chooses the next hop to route to is described next.

The paper goes into some detail describing the format of the routing table stored individually within each node in the network. The key details involve the global configuration variables b , M , and L . These values affect the size of the table and likewise the real lookup equation of $\log_{2^b}(N)$. Default values are mentioned throughout the paper and for most cases need not be modified. M represents the number of nodes stored in "neighborhood set," a set of nodes that are determined to be in local proximity to the current node. L represents the number of nodes in the "leaf set," the set of nodes that have the numerically closest nodeIds to the current node. Typical values for M and L are 2^b or 2^{b+1} . The actual routing table has $\log(N)$ rows with 2^b-1 entries in each, and is specifically set up to efficiently route over the entire range of nodeIds.

The actual routing algorithm is described in pseudocode, but it is easy to deduce. When looking for a key first a node first checks the leaf set for a guaranteed direct route, otherwise it indexes into the routing table to quickly choose the most appropriate node to send to (where the next node is the one whose prefix most closely matches searchId's prefix). The performance measures for a system with a billion nodes and default global parameters ($b=4$) is impressive. Each node had an average of 105 entries in their routing table. The average number of routing hops for any message was just 7 hops. This is clearly a very efficient and scalable routing system.

At this point in the paper the authors briefly mention portions of Pastry's API. The paper addresses the creation of nodeIds and basic message passing operations. It focuses on route, deliver, and forward methods which deals with sending, receiving, and the middleman processing of messages respectively. Pastry's simple API is designed to make the development of applications that ride on top of its overlay network easy.

The next part of the paper describes the node arrival and departure algorithms. It goes into great detail on how nodes join, how their routing table is populated from existing nodes in the system and how those existing nodes are updated with new arrivals. It was very interesting to see how efficiently the algorithm worked in practice and how quickly nodes can build an effective routing table. Briefly, a joining node, X , sends a join request to a node in the network, A , and has its nodeId calculated. The join request

gets propagated through the network to the existing node with the closest nodeId to X, let's call this node Z. Every node along the path will send their routing tables to X. Z can send its leaf set to X because they are closest in nodeId. Finally, A and X are assumed to be "neighbors." There may be some final messages between X and nodes as it finalizes its routing table, and then sends messages to nodes indicating they may make changes now that X has finally joined. The experimental results of Pastry's final joining algorithm found that approximately 14 of 15 entries per level (described later) were optimal. In comparison, Pastry's previous algorithms resulted in only 6 of 15 entries being optimal.

At the same time, the authors describe Pastry's fault tolerance for silently failing nodes. When one node can no longer communicate with another node the former assumes that the latter has failed. Here the routing table and sets should be updated to be as effective as possible to avoid sending to the non-existent node. Interestingly, Pastry makes some use of the neighborhood set in this case, as well as the more usual methods, to find a suitable replacement for the lost node. Ideally, this maintains the integrity of the network as well as the efficiency that has built up over time based on prior locality calculations. The experimental results of their node failure recovery algorithms demonstrated noticeable improvements over the naive approach when measuring the average number of message hops after a simulated failure of 10% of a 5000 node network.

One improvement Pastry provides to the rather generic routing scheme is locality metrics. The advantage is that Pastry is going to use this data to keep the same number of hops but attempt to improve the speed per hop, thus improving the overall message time by taking equally valid but technically "more optimal" routes. The paper mentions that some network proximity metric is calculated such as IP routing hops or geographic distance. They mention traceroute which could indicate raw latency calculations as well. Pastry then makes use of these metrics to put more optimal entries in the routing tables.

When building routing tables a node will collect neighborhood sets and other routing tables, and use the metrics to write its own table entries with the best possible routing characteristics. The authors mention that "the entries in the routing table of each Pastry node are chosen to be close to the present node, according to the proximity metric, among all nodes with the desired nodeId prefix." Thus producing the overall desired routing goal such that "each step moves the message closer to the destination in the nodeId space, while traveling the least possible distance in the proximity space."

As is normally the case in ring routing schemes, each step up in the routing table is exponentially increasing in terms of nodeIds. The authors describe each step as another "level" with 0 being the closest level to the current node. The paper goes on to explain how they try to determine the closest nodes, by proximity, per level of the routing table, thereby reducing message latency and network load. The experimental results showed noticeably more optimal entries, in terms of routing locality, in simulations of 10,000 nodes when compared to more naive algorithms.

At this point in the paper the ideas of fault tolerance and resistance to malicious nodes are discussed. In order to handle malicious nodes which make incorrect routing decisions Pastry makes randomized routing decisions that will eventually bypass the malicious node. They acknowledge the tradeoffs that must occur in such a scenario:

In applications where arbitrary node failures must be tolerated, the routing can be randomized. [...] In practice, the probability distribution should be biased towards the best choice to ensure low average route delay. In the event of a malicious or

failed node along the path, the query may have to be repeated several times by the client, until a route is chosen that avoids the bad node.

The authors mention one other situation where Pastry is resilient. IP routing anomalies may cause a previously successful route to fail at some point along the route (such as a router being switched off). A Pastry network may be split into disjoint parts in cases like these. This will create what is called "isolated overlays" which will individually recover on their own. Eventually, by way of periodic "expanding ring multicast search for other Pastry nodes" the two isolated overlays will detect each other and successfully merge together.

Finally, the paper discusses experimental results in their simulated Pastry networks. I provided snippets of their analysis with the appropriate sections in my analysis. However, the paper also provides insightful visual representations of their results. These are omitted here for brevity.

Analysis of Paper 2

P. Druschel and A. Rowstron, "PAST: A large-scale, persistent peer-to-peer storage utility", HotOS VIII, Schloss Elmau, Germany, May 2001.
<http://research.microsoft.com/en-us/um/people/antr/PAST/hotos.pdf>

This paper describes the design and functionality of the author's storage utility named PAST. PAST is "an Internet-based, peer-to-peer global storage utility". PAST is part of the "second generation" of decentralized, peer-to-peer routing systems that emerged after FreeNet and Gnutella. It uses the Pastry application interface to exploit Pastry's network management and routing features. PAST implements a Distributed Hash Table to expedite file-storage and replication. PAST also introduces security and storage management using trusted third-party brokers on smart cards. Additionally, it relies upon the use of hashes and public key encryption to ensure confidentiality and anonymity. The paper goes into these concepts at a high-level and relies upon the open-source PAST application itself as a reference for implementation details. The primary problems it addresses are the creation of a distributed peer-to-peer file storage vehicle, ensure reliability of information storage/retrieval, ensure fairness in storage responsibility/load, and ensure confidentiality/security of information stored in the system.

PAST relies upon Pastry's overlay network for connectivity and network management. Pastry, as also described in [1], provides the following functionality for PAST: self-organization, fault-tolerance, and $\log_{16}N$ -step boundary on network requests.

PAST's nodes provide both client access and storage. These nodes are identified by a 128-bit nodeId hashed from the node's public key. Much like RAIDS, clients select individual files to be "backed-up" into the PAST network. Likewise, files are identified by a 160-bit fileId hashed from the node's public key and the file's name. The storage and insertion requests borrows from Pastry's routing scheme. For insertions requests, the file is routed to the k nodes whose nodeId most closely matches the most significant bits of the fileId. For lookup requests, the lookup is routed to the live node whose nodeId most closely matches the significant bits of the fileId.

PAST meets its objective of information reliability in several ways. The file remains available so long as 1 of k nodes is alive. PAST relies heavily upon randomization and the probabilistic nature of its hash functions to side-step issues associated with malicious/non-functioning nodes as well as to maintain the high-availability of data. Since the hash has uniform distribution, this ensure that the file is stored on nodes that are diverse in geography, legal-rule, ownership, administration, etc. The system also

randomly requests files and checks them against their hash to ensure reliability of the file's content which in turn allows one to determine the reliability of a storage node.

PAST meets its objective of equitable storage responsibilities. The randomized routing protocol distributes system load across many nodes. The use of hashes on fileIds mathematically guarantees a uniform distribution of nodes are selected for storage across the network. The smart-card/brokerage system maintains storage-quotas through the use of storage and reclaim receipts.

PAST meets its objective of confidentiality and information security. The public-key cryptosystem and the cryptographic hashes are computationally infeasible to break. Public-key encryption on files ensures that storing nodes cannot read the contents of the files that they store. Private-key signing allows nodes to verify the originator of storage requests. Hash functions provide a simple way to verify the contents of files being stored or reclaimed. The user's identity is shielded since their nodeId is attached to a one-way hash of their public-key. The location and ownership of stored files is similarly protected since their lookup value is also on the one-way hash of the nodeId and fileId.

The authors finish by delving into related software projects. The peer-to-peer systems cited include Napster, Freenet, and Gnutella. These systems are designed around the idea of file-sharing. As such, they fail to provide the same guarantees that PAST does regarding data reliability or content persistence. OceanStore is similar to PAST but provides users with an storage system that allows for contents updates through the use of transactions. In contrast, stored files in PAST are immutable and can only ever be "reclaimed" not updated. FarSite acts as a distributed file-system that uses a directory to point users to the location of content. PAST, does not have these semantics and has opted to tightly integrate file locatability with its routing scheme. Tapestry, Chord, and CAN are referenced but are most comparable to Pastry rather than to PAST itself as far as storage functionality is concerned.

Analysis of Paper 3

Goodson, G.R.; Wylie, J.J.; Ganger, G.R.; Reiter, M.K., "Efficient Byzantine-tolerant erasure-coded storage," Dependable Systems and Networks, 2004 International Conference on , vol., no., pp. 135-144, 28 June-1 July 2004
<http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=1311884&isnumber=29105>

The main goal of a distributed backup system is to ensure that data is reliably backed up in an efficient manner. This paper discusses the author's protocol for implementing such a system. They first describe the protocol in detail, then they compare their research to a similar system that has already been developed called BFT.

The protocol works by dividing a file into "m" file chunks where "m" is an arbitrary size. Those chunks are then further split into "n" chunks where "n" is greater than "m" to increase the amount of fault tolerance that is available to the file. The larger "n" is compared to m the more fault-tolerant the system will be. In order to accomplish this first task, an erasure code is used to split the file up into "n" chunks. The code they use is a "semantic information dispersal algorithm" which stripes the data across the "m" required chunks and then creates erasure-coded fragments for the rest of the chunks. This allows any client connected to the storage system to retrieve a complete file with "m" chunks. As a result of this design, a client can retrieve a file quickly by having all of the chunks transmitted to it concurrently. It can also detect faults by receiving more than "m" chunks and validating the checksums of those pieces.

When a client would like to write to the system, the first thing that a client does is

generate a checksum for each "n" data chunk that is going to be stored on a node before it is distributed. The client will then distribute the chunks to each storage node along with a timestamp and the checksum. A storage node will only store a data chunk if the timestamp is later than the last write for that file and the checksum generated from the received data matches the checksum sent with the timestamp. If both of these conditions are true then the storage node will store the data. If an older time stamped data segment exists then the new copy will be used in its place and the old chunk will be stored until a garbage collection mechanism removes it. To protect against the client creating a byzantine fault by incorrectly generating the checksum (or from some other algorithmic error), they use a method called verifiable secret sharing, which is not elaborated upon in the paper.

In order to retrieve a file, the client creates a request and waits for all of the nodes that are storing that file to return their stored chunks to the client. The client will then look at the timestamps and checksums returned from each of the storage nodes and generate a set of candidate nodes whose data will be used for the regeneration of the original file. If the data appears to be correct the client regenerates the file and is done. If it is incomplete but repairable the client will regenerate the file then issue a write request to storage nodes to fix the problem. If the candidate set is incomplete and a file cannot be recovered the client will retry the download procedure.

The protocol developed by this paper was turned into a working system known as PASIS. The PASIS system was compared to an existing Byzantine fault tolerant system known as BFT. Comparisons were made against BFT in terms of scalability, performance, and error recovery. The results of the paper showed that the author's system was able to scale to larger number of nodes than BFT, due to lower server utilization. It could also scale to a larger number of tolerated faults, had a 60% increase in write throughput, utilized less server overhead, and had a faster response time.

Project Design

Overview:

Due to the complexity of our system, the project is split into a number of different packages each of which is responsible for a different function. The first step of storing a file in our system is dividing the file up into pieces so that it can be stored. This function is performed by the chunker package. The chunker uses a raid-5 scheme to divide the file into stripes with parity so that in the event that a piece is lost it can be recovered from the remaining pieces.

Once a file has been split, the client needs to upload the file. It does this by sending out a multicast request to nodes asking for storage space. When it receives an adequate number of responses from the clients it will need to upload the pieces. When the client needs to download the file it will have to retrieve the pieces from the storage nodes. The raids package contains the all of this functionality. The raids package also contains a number of files related to the client, terminal, and various classes to assist in the process of recovery in the event of failures. More details about the project's implementation can be found in the java docs.

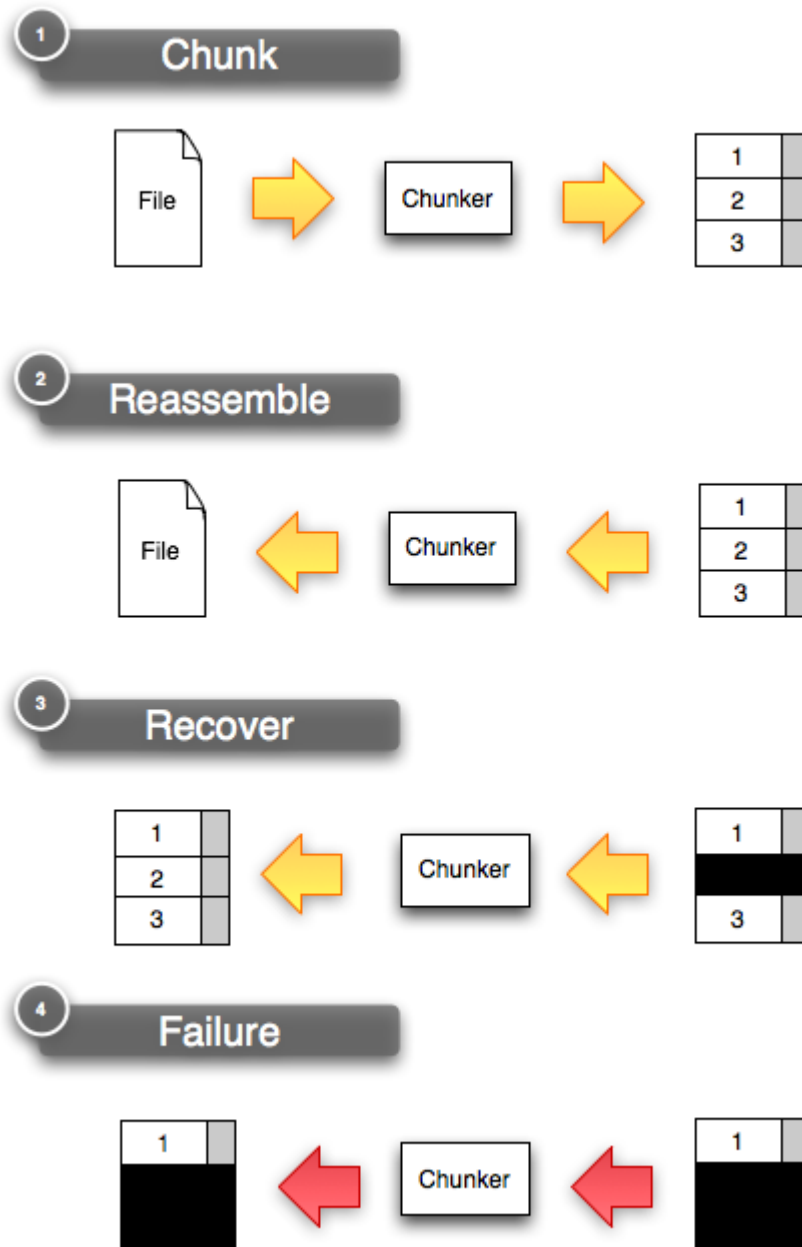
This project required us to implement several protocols for the aforementioned functionality. These protocols are the result of continual refinement to both specification and implementation. Listed here are our final designs for the protocols, all of which are implemented in the final deliverable as described.

The following is a series of functions or use cases for our program. These are in the form of a diagram for the protocol, scenario, or event followed by a description of each step in the depiction. For simplicity not everything is shown in the diagrams but enough is made available to provide the user with a high-level understanding of RAIDS's operation. While every effort was made to ensure that artifacts in the diagrams are self-explanatory, a brief legend follows:

Legend:

- circles are nodes
- solid lines are typically messages
- square blocks are typically objects or components of a system
- small blocks are typically messages.

File Splitting or Chunking:

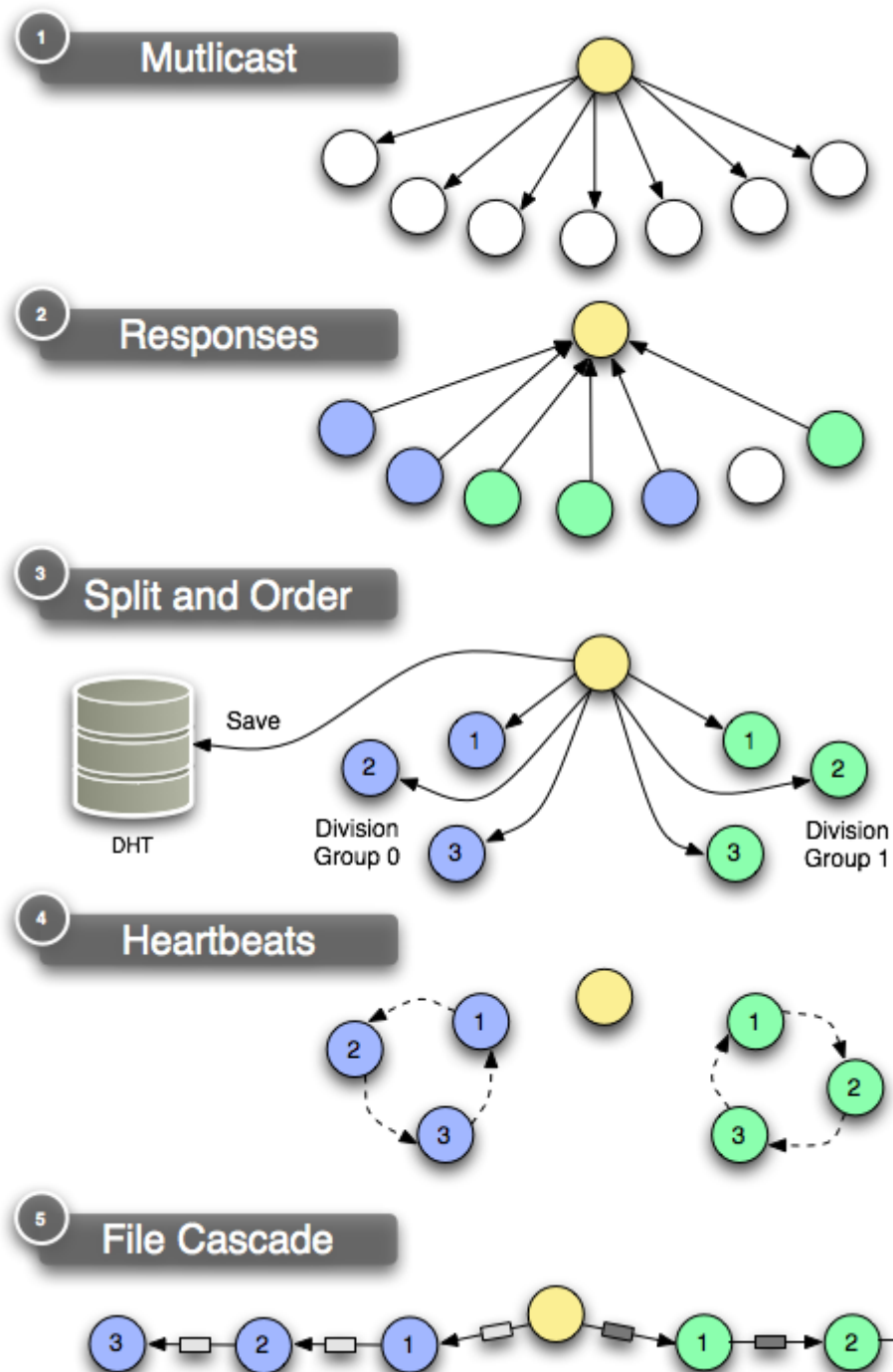


Not only is it necessary to split up a file into smaller, more manageable chunks for efficient uploading and storage over a network it is also desirable for data recovery if by chance a part gets corrupted or lost. We implemented a basic RAID like stripping algorithm to do just that. We placed the appropriate code inside the "chunker" package.

1. The Chunker can take in a file and output the desired number of chunks, with parity information. Note that because the chunks contain parity information the collective total of the output is greater than the original input. This is to be expected.
2. In normal reassembly all of the parts are available. The Chunker can then combine the parts into the original file.
3. Our implementation algorithm can reconstruct a single missing piece from all of the other pieces. This uses the parity information to reconstruct the missing stripe, and therefore recover the missing data.

4. Our implementation algorithm can only recover from a single missing piece. If more than one piece is lost then we can no longer recover and the data is permanently lost.

File Upload:

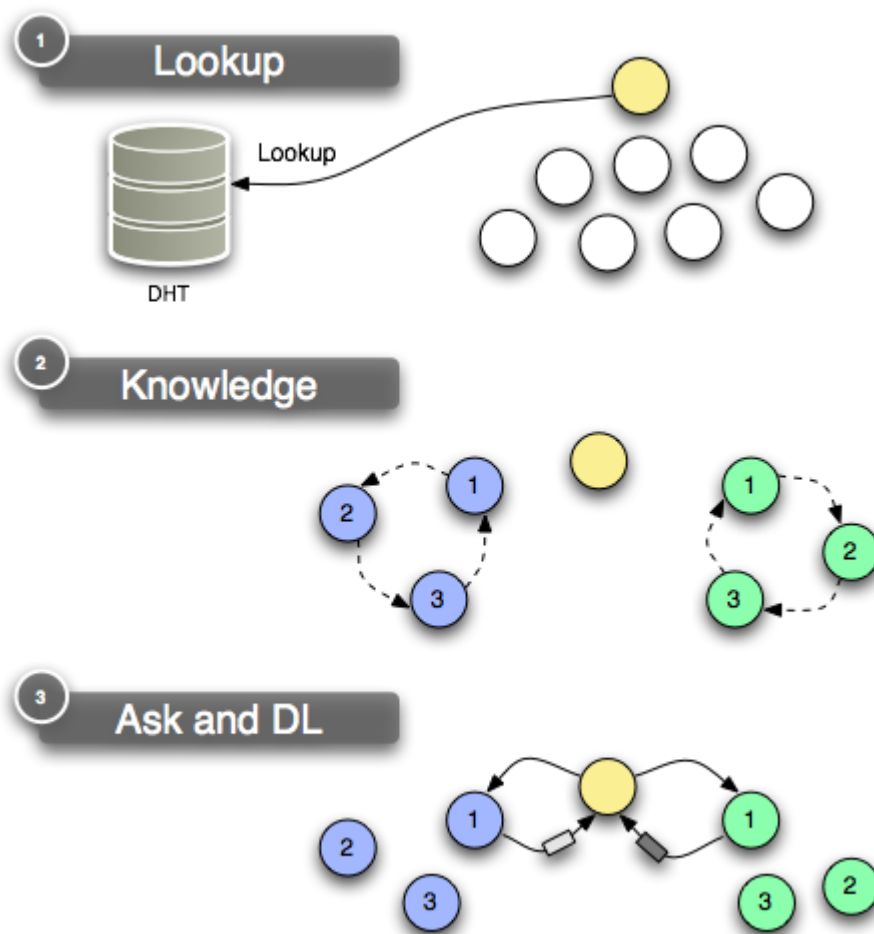


This is the core protocol in our project. The scenario starts with the yellow node, the originating client, who intends to distribute a file across the network. A precondition to this diagram is that the originating client would have successfully chunked the original file. The scenario proceeds as follows:

1. The originating client sends out a Multicast Storage Request to all of the nodes in the overlay requesting storage space. Embedded in the request is the "maximum" size the chunk could possibly be. This can be checked by the other nodes storage quota to see whether or not they can store the chunk. The request here is for 2 chunks with a replication factor of 3, thus the originating client is looking for a total of 6 nodes.
2. At least 6 nodes respond to the storage request claiming they have available space and are willing to store the chunk. The originating client takes only the first 6 responses and ignores all of the rest. From this point on these are the only nodes we will be concerned with.
3. With the list of 6 willing nodes the originating client can arbitrarily split the nodes into groups. This becomes the "Master List" describing all the nodes, the parts they should contain, and their ordering, and more meta information. This Master List is stored in the DHT under a specific lookupId (also contained in the Master List itself) and sent directly to all of the concerning nodes. There is also another value stored in the DHT. This is the "PersonalFileList" for the user on the originating client. This way, the user can join the peer-to-peer network from anywhere, and by fetching his PersonalFileList from the DHT will know that he has uploaded this file, and what to expect. However, this has little to do with the uploading protocol itself.
4. Upon receiving the "MasterListMessage" the nodes form their heartbeat rings. This ring formation is significant in a number of ways. First and foremost it is efficient. For a ring of any size a member of the ring only needs to send heartbeats to one other node and listen for heartbeats from one other node. Secondly there is an explicit ordering that is beneficial in the recovery process for fault tolerance. The ordering makes it explicit as to which node is tasked with acting in the recovery protocol. This is shown more in the Recovery diagram.
5. Finally the originating client *pushes* the appropriate chunk to the first node in each division group. Upon completing the download those nodes will then push the chunk to the next node in the ring. We refer to this process as "cascading" the chunk. Work can be done in this area to more efficiently distribute the bytes rather than one node at a time, however this accomplishes our original goal of ensuring that the originating client only uploads no more than 100% of the bytes of the chunked file. Anything more would be nice, but unnecessary. Once the file has cascaded around the ring then every node has the file chunk and is maintaining their keep-alive heartbeats in the ring. This is an ideal situation.

It is worth pointing out that there is a potential race condition if the file chunk is *pushed* to a node before it has received the MasterList telling that node that it is to store that part. We are aware of this race condition and handle it gracefully. We keep the file and store it in our inventory anyways. A more advanced algorithm might be to allow the file to expire after a certain amount of time.

File Download:



The download scenario is much simpler as a result of the precautions and data structures put in place during the upload scenario. Again the yellow node is the originating client, with the same username as the one that uploaded before, and they intend to download the same file they originally uploaded.

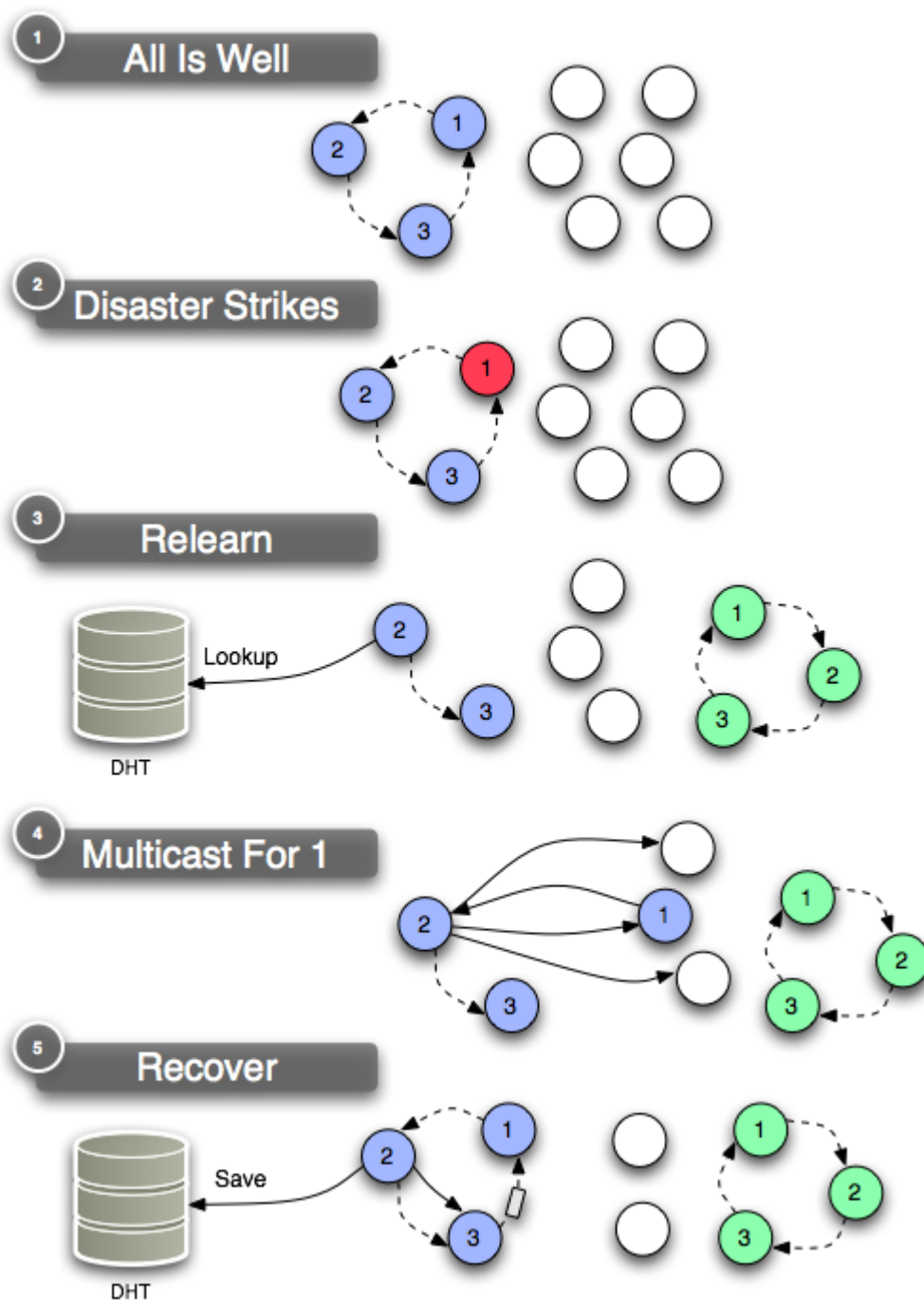
1. When this scenario starts the originating client knows nothing about the existing peer-to-peer network. They may have left the network and rejoined later, or they may even still be running since the time they uploaded. However, the first step is always to lookup the up-to-date "Master List" in the Distributed Hash Table (DHT). The Master List contained in the DHT is the definitive reference for the current state of the nodes, containing which parts, etc. This will make sense when you see later that during node recovery this list is updated to reflect the updates.
2. Once the Master List has been fetched from the DHT the originating client has all of the knowledge that it needs to visualize the division groups.
3. The originating client can then arbitrarily choose nodes from each division group to download the parts from. It sends the node a "DownloadMessage" and that node then pushes the file back to the originating client. Verification measures are taken and reassembly of the parts will result in the original file.

During the download stage it is possible, through node failure, that a node in the division group ring *does not yet* contain the file. In such a case the file, due to our recovery algorithm discussed next, that node will eventually receive the file or may be downloading it at that moment. In either case they can send back an "empty file" to the originating client to indicate they don't have the file. The originating client can then

cross that node off of its list and attempt to download from the next node in that division group.

Finally, still one more bad scenario may exist. If all of the nodes in a division group vanish at once then Fault Tolerance within that group will be non-existent. Effectively, that file chunk is permanently lost unless it is rebuilt from the other pieces. In such a case, the originating client will attempt to download from the now non-existent nodes in that division group, it will receive timeouts from each and exhaust that list of nodes and report the file as missing. Reassembly is still possible due to the parity of data on the other chunks! So the download continues and should be successful!

Fault Tolerance:



Fault Tolerance in our system is sufficient enough to recover from normal node failures and the like. However, we do not attempt to repair from any catastrophic events such as the failure of a complete division group, or the DHT itself.

The above scenario shows the division group 0 (the blue nodes) recovering from the silent failure of one of the nodes at right (blue node 1).

1. This would be a normal state of the peer-to-peer network under ideal conditions. Blue node 2 receives heartbeat messages (thumps) from 1 and sends thumps to 3. All is well.
2. A feature of heartbeat ring is that each particular node has known responsibilities. In this case, when the node fails the thumps it was sending to blue node 2 will no longer be sent and 2 will detect it. At this point node 2, the

node listening to a node, is tasked with jump starting the recovery algorithm.

Likewise, blue node 3 will have his thumps timeout, and will do nothing other than stop sending thumps to that node and update his internal structures accordingly.

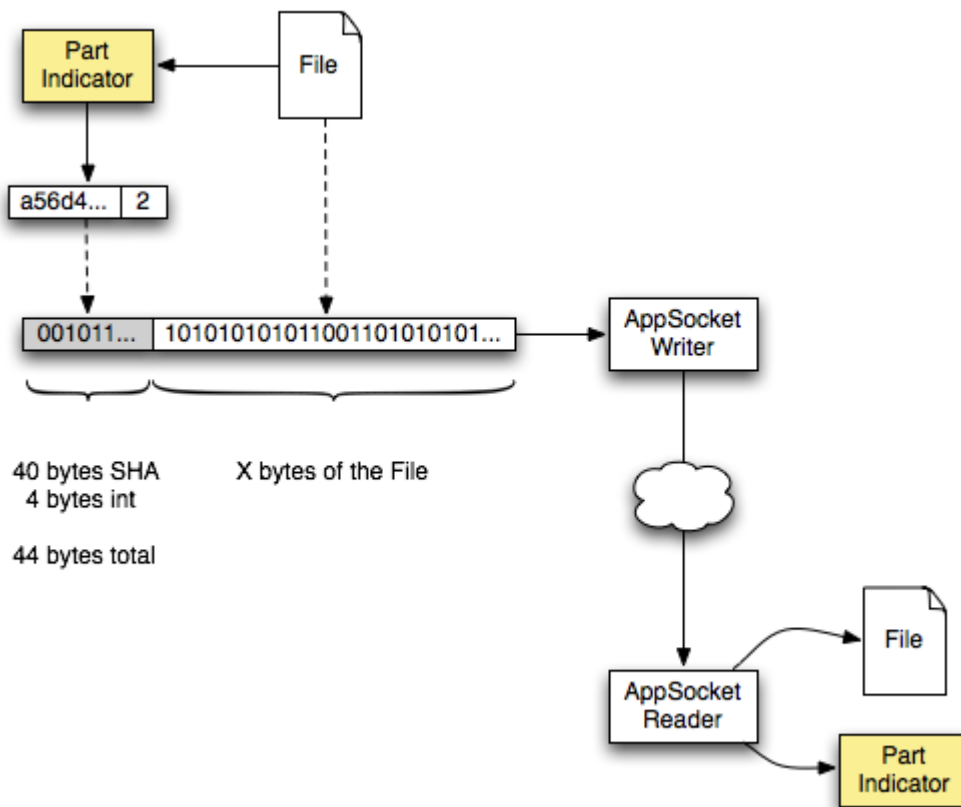
3. Like with the download scenario the most up to date Master List should be fetched from the DHT. This knowledge allows node 2 (the medic) to know all the other nodes storing parts for this file. The medic can use this information to its advantage in the next step.
4. The medic must find a replacement for the failed node. At this point it can multicast, excluding the nodes already storing parts, and find a node willing to store the part. The request here is for a single node to take the place of the failed node. Thus, the medic can ignore all but the first response.
5. The medic updates the Master List and stores that in the DHT. This ensures that the DHT always has the most up-to-date listing of the network. It is possible that every single node may have changed from the original update to the first download request, but as long as the DHT has been updated it will make no difference to the originating client. Now, in order to repair the ring the medic must send the "MasterListMessage" of data he just stored in the DHT to the newly joining node (not pictured). Then, after calculating the node prior to the new node in the ring (in this case blue node 3) it sends a "RecoverMessage" to that node. The RecoverMessage tells blue node 3 who it should start sending heartbeats to and also to cascade the file chunk to that node. At this point everything should be up to date exactly like the ideal situation.

The ring principle allows us to exploit the "cascade" idea in fault tolerance. If all but a single node in the ring dies then the system *can* recover. Each node will repair the node prior to it, and that one the node prior, until the ring is completely reconstructed backwards, and the file will cascade around it!

There are a few scenarios our implementation does not handle. In this example there is a race condition between fetching and updating the DHT records. If a node in division group 0 fails and the recovery starts, followed by a node in division group 1 failing and the recovery starting before the other recovery writes a new record then there will be a lost write to the DHT. We do not explicitly handle this. We could somehow lock the DHT record or merge the two writes however we chose not to handle this. In normal usage this would be extremely rare and even if it were to happen the system is resilient enough that it will still function and may eventually recover on its own.

There are still more edge cases that our Fault Tolerance does not handle. We felt that handling some of these more obscure scenarios was outside the scope of this project and would require more time than the assignment allotted.

AppSockets - File Transfers:



File Transfers are handled using FreePastry's "AppSocket" interface. This is required because FreePastry has a single Network thread. This single thread handles all message passing, including FreePastry's own network maintenance messages. If we were to send a large file in a message it would block the network thread and cause nodes to think the transferring nodes died because they aren't responding. FreePastry provides their AppSocket as a framework for sending larger messages, such as files, instead of requiring that we implement our own solution with TCP Sockets. AppSockets send and receive ByteBuffers.

An important point is that in many different places in our system we use a "PartIndicator" data structure to refer to a particular chunk of a particular file. We use this data structure, instead of just the SHA1 hash of that chunk, because we give special meaning to the SHA1 hash of a chunk. Our design reserves the SHA1 hash of a chunk as the verification and trust building communication for nodes in a division group ring holding that chunk. Therefore, if it is used for verification it is kept hidden at all times. The PartIndicator has the exact same semantic meaning as the SHA1 hash for the part would have. The hash cannot be inferred from the PartIndicator, a very generic structure, so it is used instead as a safe way to refer to a specific file chunk.

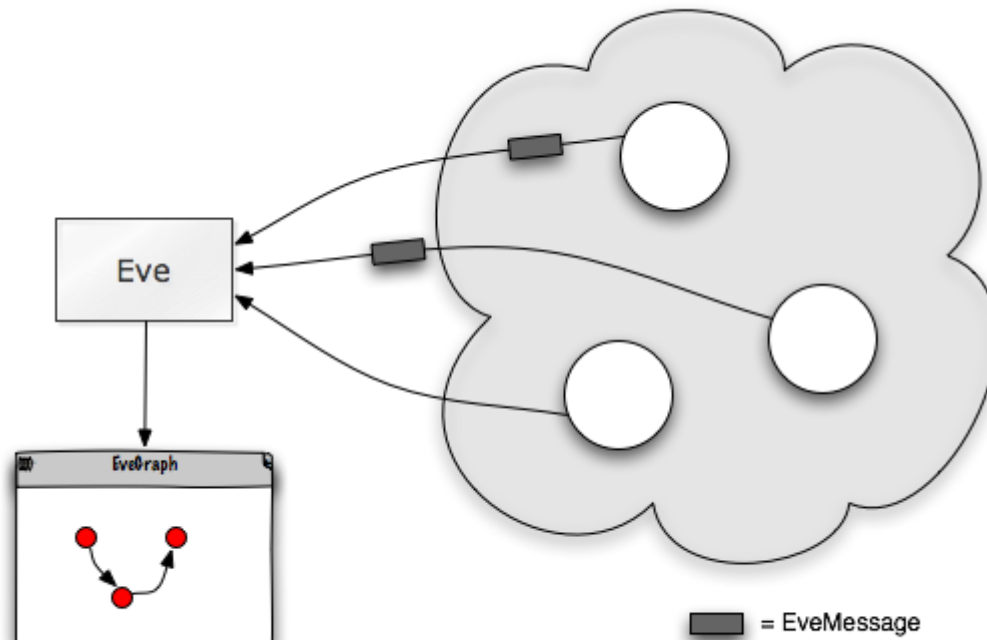
Therefore whenever we send a file over the wire, through a byte buffer, we first prepend the PartIndicator describing that piece. The PartIndicator can be represented in 44 bytes. 40

describing the SHA1 String as ASCII Hex (rather than the 20 raw bytes) and 4 bytes representing the ID. Following this "header" is the raw file data.

Our "BufferUtils" class properly handles assembling the buffers for sending over the wire, and we have setup JUnit tests to verify if anything changes that it will be caught quickly. The buffer is read and assembled into both the raw file (a tempfile) and the

PartIndicator for that file. These two objects are handed off to the Raid Application (the node) to be handled appropriately.

Eve Visualization:



In order to conceptualize the complex peer-to-peer network, follow message passing scenarios, and verify that our protocol was working we chose to build a simple graph visualizer. We called this visualizer "Eve" and the source code is contained inside the "eve" package.

There are two optional command line parameters to the Client. They are for the Eve host and port. If provided, and if Eve is running then each node started in that Client java process will build an EveReporter that is attached to that Eve instance. EveReporter is essentially just a Logging interface where the node can send messages when it wants to. Those messages go to Eve, which synchronizes them and displays them appropriately on the Graph.

Some advantages of this approach are that you can pick and choose the nodes that you want to appear in the Eve visualizer. Because you can specify whether or not you want to connect to eve on the command line, you can specify some to connect and others to not. Eve itself is smart enough to handle creating a vertex if a message refers to a node that it did not know about. This simple visualization made it possible to diagnose some early message passing logic in our code and was very useful for demonstrations.

User's Manual

Command Line - Preferred: (Mac / Unix / Linux / Solaris)

To get the source code you can either extract the archive provided in the deliverable or checkout the source code from the Subversion Repository. To extract from a jar archive or checkout a read only copy of our project you can run the following commands:

```
shell> jar xf monotonicity.jar
shell> cd monotonicity
```

or

```
shell> svn checkout http://rit-monotonicity.googlecode.com/svn/trunk/ monotonicity
shell> cd monotonicity
```

Once extracted or checked out you will have to move into the new directory and run some configuration files to setup the proper Java Classpath. We have provided a shell script to set-up the global \$CLASSPATH variable on a Unix/Linux machine to include the appropriate jars and class files to run our project. Execute the following from the project root directory (you should be here from the previous step):

```
shell> source shell/configuration.sh
```

This command sets up your classpath, builds the java class files, generates the javadocs, and produces a helpful usage message. Note that you will only have to run the configuration.sh file once. If you want to setup your classpath for future shells, and not have to recompile, use the following command:

```
shell> source shell/classpath.sh
```

You should now be all setup to run commands from the command line.

Eclipse - For Development or Read Access: (Any Platform)

If you have Eclipse setup with a Subversion plugin it may be easier to checkout our project directly into Eclipse. The following a tutorial for the Subclipse plugin available at <http://subclipse.tigris.org>:

1. Select File → New → Other → Checkout Projects from SVN
2. Select "Create a new repository location" and click "Next"
3. Input this URL: <http://rit-monotonicity.googlecode.com/svn/trunk/> and Click "Next"
4. In the resulting list for that location click on the root repository icon and Click Next.
5. Select "Java Project" (this is important).
6. Give the Project a unique name and complete the setup wizard.

Thats it for checking out the source. To run application you can observe the provided command line arguments for the shell version and convert them to the appropriate run configurations for Eclipse.

How to Run

I would recommend opening two shells. One to run Eve and the other to run the main Client. However, it doesn't have to be done this way, that is just how it will displayed below. The first shell will run Eve:

```
shell> source shell/classpath.sh    # Setup the Classpath just in case
shell> java eve.Eve 9999            # Listening on port 9999
```

Starting up Eve will initialize a GUI. Therefore it shouldn't be done unless you have windowing capability or java will complain that there is no capability. Once Eve is started you can run a Client:

```
shell> source shell/classpath.sh    # Setup the Classpath just in case
shell> java raids.Client 9000 localhost 9000 axel 10 localhost 9999
```

This will start up the Client application. You will get a usage message explaining the command line arguments if you type them in wrong:

```
shell> java raids.Client
Too Few Command Line Arguments
example: java raids.Client 9000 localhost 9000 joe 10 localhost 9999
usage: java Client listenPort bootIP bootPort username numNodes [eve_host
eve_port]
    listenPort = port this node will listen on
    bootIP     = bootstrap node's IP
    bootPort   = bootstrap node's port
    username   = the Client's username
    numNodes   = number of nodes to create on this JVM
    eve_host   = optional override EVE_HOST property
    eve_port   = optional override EVE_PORT property
```

The first three arguments are setup for the FreePastry network. The first port number is the port the node you created will start listening on. Also, if you create multiple nodes then the port number will increment by one for each additional node. The next two arguments are the bootstrap node's IP and port. If you wish to connect to a node across the network (this is possible) you would have to provide the IP address and port number that the existing node is listening on. For testing multiple nodes on your own machine "localhost" is easiest.

The username parameter is specific to our application. As long as the Pastry network stays active then you can rejoin, with your username, from any machine, and maintain your list of uploaded files. If the number of nodes created is 1 then the username will match exactly as the specified on the command line. If the number of nodes created is n , such that $n > 1$, then the username will be suffixed with a counter from 0 to $(n-1)$.

The parameter after the username specifies the number of nodes to create on this virtual machine. 10 is a reasonable number for testing purposes on one machine, but we have tested with up to 150 nodes, about the point where Java claims there are "Too many open files." However, the more nodes there are the harder it is to visualize and understand what is going on.

The last two parameters are optional. If you are running Eve you can provide the Client with Eve's host and port number so that the nodes log their actions with Eve. You can therefore have some nodes that connect to Eve and others that do not as long as you spawn them with separate Client processes and control their command line arguments.

Once the Client is started you will see the nodes being created followed by a prompt. Some of the debug output was left in so that an idea about what is happening behind the scenes can be gleaned. There is also some output from FreePastry itself that we have no control over that you may have to ignore. Any potentially bad looking output from FreePastry itself is likely to be expected output! Unless an exception trace appears to come from our source code (raids.* package) then it's something we could not easily control.

Once presented with the prompt from our Client application you can type "help" to get a list of the commands you can use to interact with the peer-to-peer network.

```
shell> java raids.Client 9000 localhost 9000 axel 10 localhost 9999
Finished creating new node PastryNode[...]
Finished creating new node PastryNode[...]
Finished creating new node PastryNode[...]
...
>> help
```

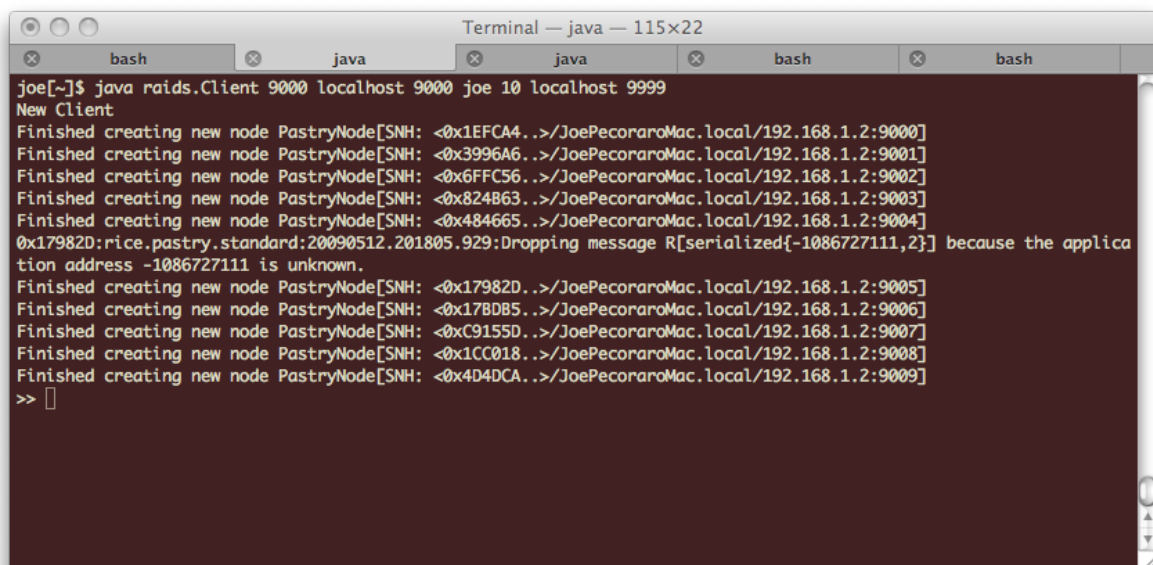
Commands:

help	This help menu
list	Prints the status information of all nodes on this JVM
kill [#]	Kills the given node, or the current if none is given
status	Prints status information on the current node
switch #	Switches to the given node
upload path #	Chunks and uploads a file to 3*# nodes in the network
download filename	Downloads the file uploaded by this user with that name
quit	Exits the client program

The next section shows an entire sample run through.

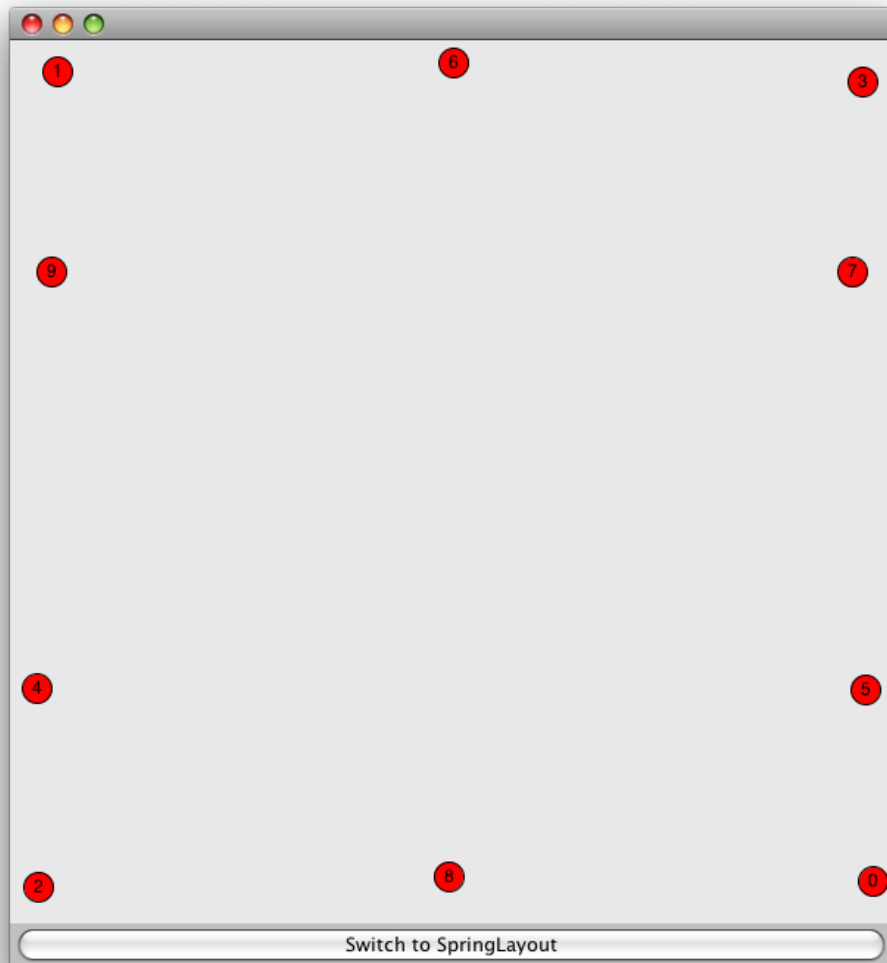
Walkthrough

I will walk through an entire test scenario, showing all of the commands, explaining the process and interpreting the results. I start by executing the same commands as above to launch Eve, and then I launch a Client with 10 nodes:



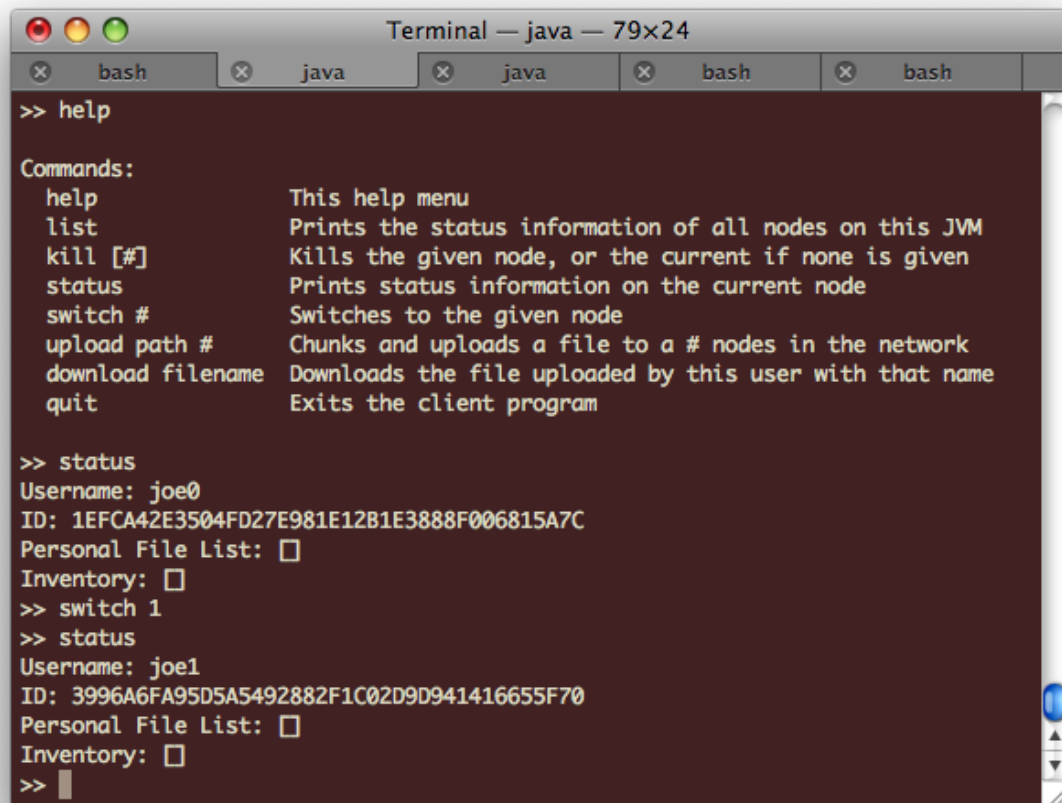
```
Terminal — java — 115x22
joe[~]$ java raids.Client 9000 localhost 9000 joe 10 localhost 9999
New Client
Finished creating new node PastryNode[SNH: <0x1EFC44..>/JoePecoraroMac.local/192.168.1.2:9000]
Finished creating new node PastryNode[SNH: <0x3996A6..>/JoePecoraroMac.local/192.168.1.2:9001]
Finished creating new node PastryNode[SNH: <0x6FFC56..>/JoePecoraroMac.local/192.168.1.2:9002]
Finished creating new node PastryNode[SNH: <0x824863..>/JoePecoraroMac.local/192.168.1.2:9003]
Finished creating new node PastryNode[SNH: <0x484665..>/JoePecoraroMac.local/192.168.1.2:9004]
0x17982D:rice.pastry.standard:20090512.201805.929:Dropping message R[serialized{-1086727111,2}] because the applica
tion address -1086727111 is unknown.
Finished creating new node PastryNode[SNH: <0x17982D..>/JoePecoraroMac.local/192.168.1.2:9005]
Finished creating new node PastryNode[SNH: <0x17BD85..>/JoePecoraroMac.local/192.168.1.2:9006]
Finished creating new node PastryNode[SNH: <0xC9155D..>/JoePecoraroMac.local/192.168.1.2:9007]
Finished creating new node PastryNode[SNH: <0x1CC018..>/JoePecoraroMac.local/192.168.1.2:9008]
Finished creating new node PastryNode[SNH: <0x4D4DCA..>/JoePecoraroMac.local/192.168.1.2:9009]
>> []
```

The command line produces 10 nodes. Their names are actually joe0 through joe9 where the suffix number matches the port number of the created nodes. So joe0 is the node on port 9000 and joe9 is the node on port 9009. This will be helpful in analysis later. You can see the FreePastry output, about dropping a message, which we cannot control. This causes no harm. After spawning the nodes they registered with Eve and you will see vertexes appear on the Eve visualizer:



Once again this has been setup so that the node with username "joe3" is vertex 3 in the Eve visualizer. This property only holds for the original Client connecting to Eve and not for future Clients connecting to the same Eve instance.

The terminal allows you to jump between the nodes spawned by the Client. The following example shows you can check out information on the current node you are attached to with ``status`` and you can switch to another node by using the ``switch`` command.

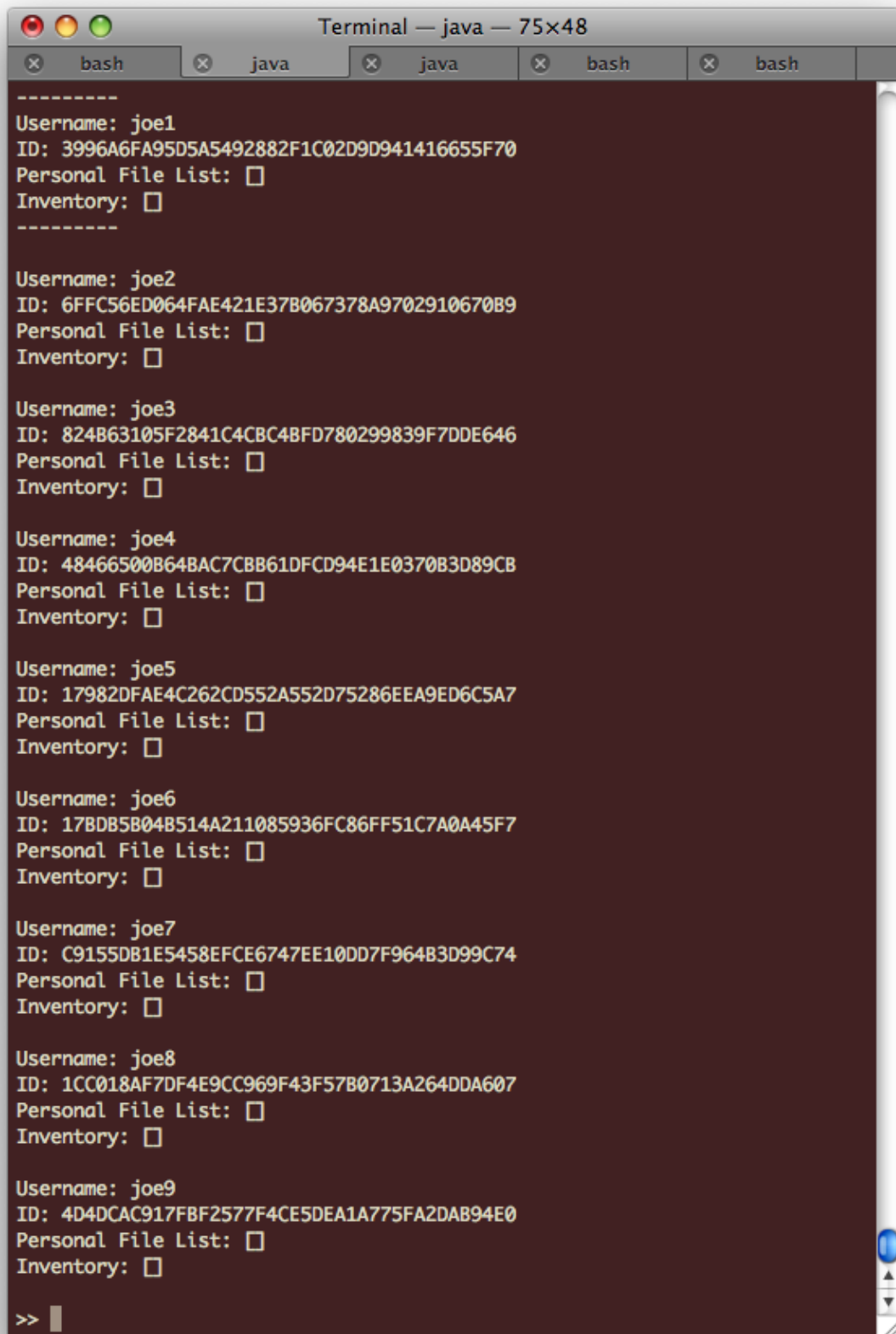
A screenshot of a macOS Terminal window titled "Terminal — java — 79x24". The window has several tabs at the top: "bash", "java", "java", "bash", and "bash". The first "java" tab is active. The terminal content shows a help menu with commands like "help", "list", "kill [#]", "status", "switch #", "upload path #", "download filename", and "quit". The user enters "help" and "status". The "status" command outputs the username "joe0", a long SHA1 node ID, and empty lists for "Personal File List" and "Inventory". The user then enters "switch 1", and the next "status" command shows the username changed to "joe1" and a new node ID.

```
Terminal — java — 79x24
bash java java bash bash
>> help

Commands:
  help          This help menu
  list          Prints the status information of all nodes on this JVM
  kill [#]      Kills the given node, or the current if none is given
  status        Prints status information on the current node
  switch #      Switches to the given node
  upload path # Chunks and uploads a file to a # nodes in the network
  download filename Downloads the file uploaded by this user with that name
  quit          Exits the client program

>> status
Username: joe0
ID: 1EFC442E3504FD27E981E12B1E3888F006815A7C
Personal File List: []
Inventory: []
>> switch 1
>> status
Username: joe1
ID: 3996A6FA95D5A5492882F1C02D9D941416655F70
Personal File List: []
Inventory: []
>> 
```

The status shows that I started on node "joe0", I then switched to "joe1" with the `switch 1` command. Notice that the status command prints out more information than just the username of the node. It prints out the SHA1 nodeId, the list of files uploaded by the user and the list of chunks stored in the node's inventory. You can get a general listing of all the nodes with the `list` command:

A terminal window titled "Terminal — java — 75x48" with tabs for "bash", "java", "java", "bash", and "bash". The terminal displays a list of users and their IDs, with the first user highlighted. The output is as follows:

```
-----
Username: joe1
ID: 3996A6FA95D5A5492882F1C02D9D941416655F70
Personal File List: ☐
Inventory: ☐
-----

Username: joe2
ID: 6FFC56ED064FAE421E37B067378A9702910670B9
Personal File List: ☐
Inventory: ☐

Username: joe3
ID: 824863105F2841C4CBC48FD780299839F7DDE646
Personal File List: ☐
Inventory: ☐

Username: joe4
ID: 48466500B64BAC7CBB61DFCD94E1E0370B3D89CB
Personal File List: ☐
Inventory: ☐

Username: joe5
ID: 17982DFAE4C262CD552A552D75286EEA9ED6C5A7
Personal File List: ☐
Inventory: ☐

Username: joe6
ID: 17BDB5B04B514A211085936FC86FF51C7A0A45F7
Personal File List: ☐
Inventory: ☐

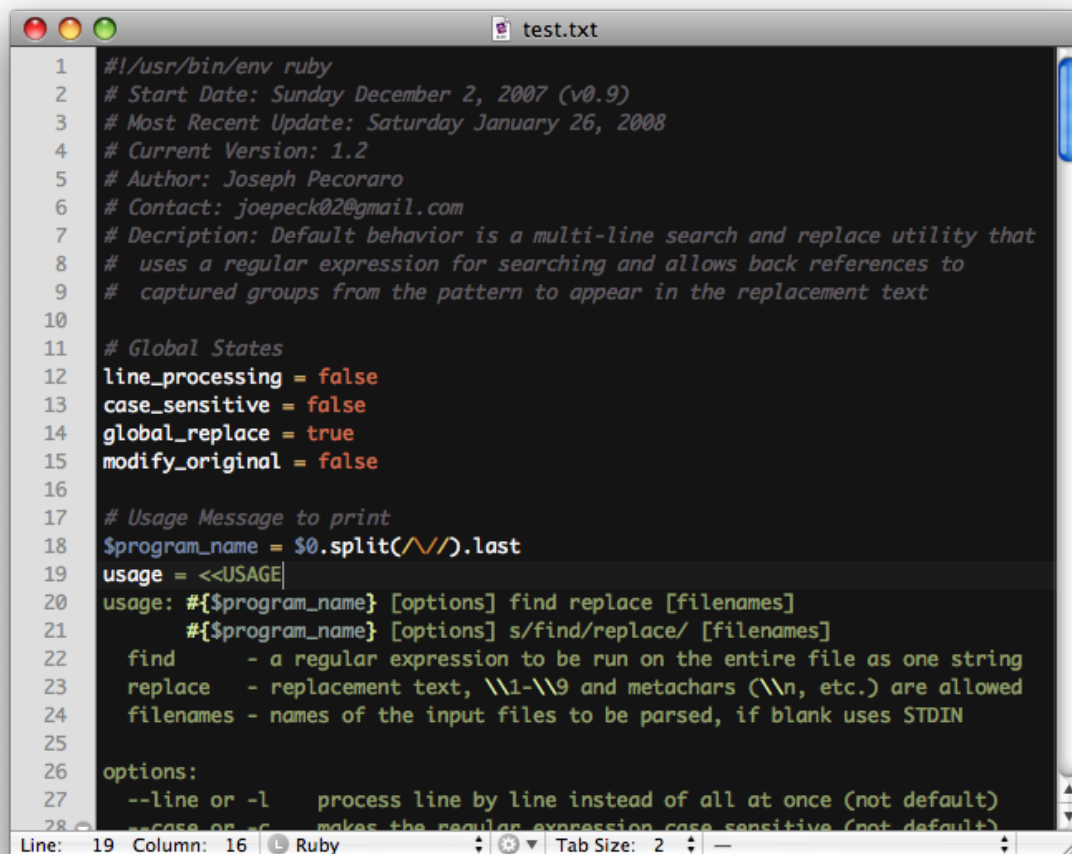
Username: joe7
ID: C9155DB1E5458EFCE6747EE10DD7F964B3D99C74
Personal File List: ☐
Inventory: ☐

Username: joe8
ID: 1CC018AF7DF4E9CC969F43F57B0713A264DDA607
Personal File List: ☐
Inventory: ☐

Username: joe9
ID: 4D4DCAC917FBF2577F4CE5DEA1A775FA2DAB94E0
Personal File List: ☐
Inventory: ☐

>> |
```

Notice that the node you are currently on is highlighted slightly with bars around it. This view is useful to get a quick view of the overall system. Next we will upload a file. The upload command requires the absolute path to a file on the system. The file that I will be using has absolute path `"/tmp/test.txt"` and contains some text:

A screenshot of a text editor window titled 'test.txt'. The window has a dark background with light-colored text. The text is a Ruby script. The first 10 lines are comments in grey. Lines 11-15 are variable assignments. Lines 16-17 are comments. Line 18 is a variable assignment. Line 19 is a comment. Lines 20-24 are a multi-line comment block. Line 25 is a comment. Line 26 is a comment. Line 27 is a comment. Line 28 is a comment. The status bar at the bottom shows 'Line: 19 Column: 16 Ruby' and 'Tab Size: 2'.

```
1  #!/usr/bin/env ruby
2  # Start Date: Sunday December 2, 2007 (v0.9)
3  # Most Recent Update: Saturday January 26, 2008
4  # Current Version: 1.2
5  # Author: Joseph Pecoraro
6  # Contact: joepeck02@gmail.com
7  # Description: Default behavior is a multi-line search and replace utility that
8  # uses a regular expression for searching and allows back references to
9  # captured groups from the pattern to appear in the replacement text
10
11 # Global States
12 line_processing = false
13 case_sensitive = false
14 global_replace = true
15 modify_original = false
16
17 # Usage Message to print
18 $program_name = $0.split(/\/\//).last
19 usage = <<USAGE|
20 usage: #{ $program_name } [options] find replace [filenames]
21       #{ $program_name } [options] s/find/replace/ [filenames]
22       find      - a regular expression to be run on the entire file as one string
23       replace   - replacement text, \\1-\\9 and metachars (\\n, etc.) are allowed
24       filenames - names of the input files to be parsed, if blank uses STDIN
25
26 options:
27 --line or -l    process line by line instead of all at once (not default)
28 --case or -c    makes the regular expression case sensitive (not default)
```

The upload parameters I provide are "/tmp/test.txt" which is the above file and "2" to split the file into 2 chunks. The replication factor is currently hardcoded in the system at 3, because this ensures nice properties to display. However the replication factor can be changed to any reasonable value, 2-5 and the system works as expected.

So, by providing these values the expected results are that 3 nodes should respond for each of the 2 chunks, thus giving us a total of 6 nodes to work with. I am issuing the command from "joe1" so vertex 1 is the originating client and will set things in motion. Here are the results:


```
Terminal — java — 100x46
x bash x java x java x bash
>> upload /tmp/test.txt 2
Replication Factor: 3

Nodes that Responded with Storage Space: (6)
[SNH: <0x4D4DCA..> //192.168.1.2:9009]
[SNH: <0x17BDB5..> //192.168.1.2:9006]
[SNH: <0x1CC018..> //192.168.1.2:9008]
[SNH: <0xC9155D..> //192.168.1.2:9007]
[SNH: <0x1EFCA4..> //192.168.1.2:9000]
[SNH: <0x17982D..> //192.168.1.2:9005]

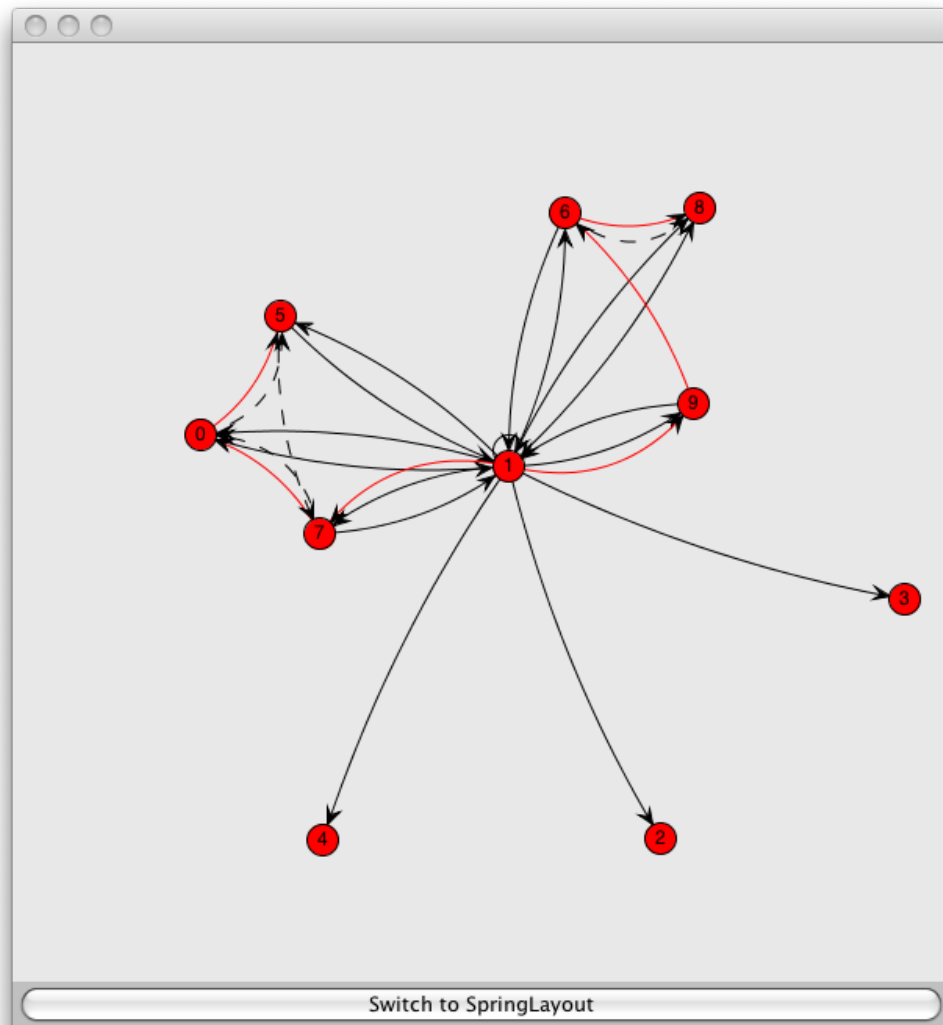
Division Group (0):
[SNH: <0x4D4DCA..> //192.168.1.2:9009]
[SNH: <0x17BDB5..> //192.168.1.2:9006]
[SNH: <0x1CC018..> //192.168.1.2:9008]

Division Group (1):
[SNH: <0xC9155D..> //192.168.1.2:9007]
[SNH: <0x1EFCA4..> //192.168.1.2:9000]
[SNH: <0x17982D..> //192.168.1.2:9005]

SHA HASH: a50e7ceb6c2f72549b340fd4d28ef4b7db2b07a8
Sending 81E3063ADFB2F58D8692249DEE19F5BB006E022:0 to 4D4DCAC917FBF2577F4CE5DEA1A775FA2DAB94E0
Sending 81E3063ADFB2F58D8692249DEE19F5BB006E022:1 to C9155DB1E5458EFCE6747EE10DD7F96483D99C74
Successfully Submitted the File, it will be uploading in the background.
>> Successfully stored the MasterListContent for <0x81E306..> at 2 locations.
Successfully stored PersonalFileList for joe1 at 2 locations.
Dumped the 4052 bytes from the buffer into the temp file >> /tmp/RAIDS-appsocketreader-20362.tmp
Dumped the 4052 bytes from the buffer into the temp file >> /tmp/RAIDS-appsocketreader-20361.tmp
Dumped the 1520 bytes from the buffer into the temp file >> /tmp/RAIDS-appsocketreader-20361.tmp
Dumped the 1520 bytes from the buffer into the temp file >> /tmp/RAIDS-appsocketreader-20362.tmp
4D4DCAC917FBF2577F4CE5DEA1A775FA2DAB94E0: Received part: 81E3063ADFB2F58D8692249DEE19F5BB006E022:0
C9155DB1E5458EFCE6747EE10DD7F96483D99C74: Received part: 81E3063ADFB2F58D8692249DEE19F5BB006E022:1
Dumped the 4052 bytes from the buffer into the temp file >> /tmp/RAIDS-appsocketreader-20363.tmp
Dumped the 4052 bytes from the buffer into the temp file >> /tmp/RAIDS-appsocketreader-20364.tmp
Dumped the 1520 bytes from the buffer into the temp file >> /tmp/RAIDS-appsocketreader-20364.tmp
Dumped the 1520 bytes from the buffer into the temp file >> /tmp/RAIDS-appsocketreader-20363.tmp
1EFCA42E3504FD27E981E12B1E3888F006815A7C: Received part: 81E3063ADFB2F58D8692249DEE19F5BB006E022:1
17BDB5B048514A211085936FC86FF51C7A0A45F7: Received part: 81E3063ADFB2F58D8692249DEE19F5BB006E022:0
Dumped the 4052 bytes from the buffer into the temp file >> /tmp/RAIDS-appsocketreader-20365.tmp
Dumped the 1520 bytes from the buffer into the temp file >> /tmp/RAIDS-appsocketreader-20365.tmp
1CC018AF7DF4E9CC969F43F57B0713A264DDA607: Received part: 81E3063ADFB2F58D8692249DEE19F5BB006E022:0
Dumped the 4052 bytes from the buffer into the temp file >> /tmp/RAIDS-appsocketreader-20366.tmp
Dumped the 1520 bytes from the buffer into the temp file >> /tmp/RAIDS-appsocketreader-20366.tmp
17982DFAE4C262CD552A552D75286EEA9ED6C5A7: Received part: 81E3063ADFB2F58D8692249DEE19F5BB006E022:1
```

The clean output at the top is normal output. The output at the bottom is debug output that I will explain in detail. First, joe1 issues the multicast, therefore sending a message to all the nodes including himself. He then successfully got 6 responses to the multicast storage request, and they are nodes joe9, 6, 8, 7, 0, and 5. They are split into Division Groups for each of the 2 chunks. Division Group 0 = [9,6,8]. Division Group 1 = [7,0,5]. Note the ordering is important! That is the ordering of the ring mentioned earlier in this document. The originating client uploads the appropriate chunk to the first node in each division, and they then cascade the file to the next, and the next, completing the ring.

Indeed that is precisely what the visualizer shows:



The debug output highlights some important details that happened in the background.

Multiple values were stored in the Distributed Hash Table (DHT). Both the Master List for this file and the Personal File List for this user were stored in the DHT and replicated in two locations. Remember that the DHT is distributed across the nodes pictured above. We make use of FreePastry's PAST implementation to handle those messages so we don't display them in Eve, only our own.

The rest of the debug shows the file transfers and where they are actually being stored on the machines. There are some very minute details here. The AppSockets are reading in 4kb sections, and the first section sent over the wire always contains the 44 bytes representing the PartIndicator object. That means that the following data is getting sent over the wire:

44 bytes for the PartIndicator (4kb - 44 = 4052 left over for raw data)
4052 bytes raw data + 1520 bytes raw data = 5572 bytes of raw data

That is exactly what the system indicates:

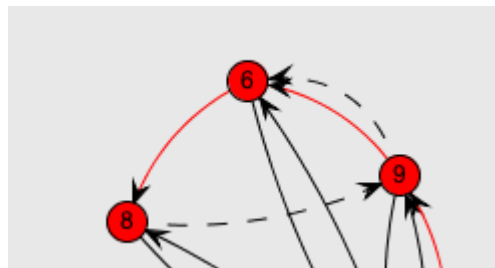
```
Terminal — bash — 85x7
joe[~]$ ls -l /tmp/RAIDS-appsocketreader-20362.tmp
-rw-r--r-- 1 joe wheel 5572 May 12 20:20 /tmp/RAIDS-appsocketreader-20362.tmp
joe[~]$
```

You can also see that the PartIndicator's string representation is a SHA1:# for example:

```
81E3063ADFB2F58D8692249DEE19F5BB006E022:0
81E3063ADFB2F58D8692249DEE19F5BB006E022:1
```

That SHA1 hash is the lookupId you would use to find the value in the DHT. The part number is the chunk number of that file. This way you can find out anything you need to know, except the verifying hash of that chunk's contents, using the PartIndicator.

Also visible in the Visualizer are the Heartbeat message. They appear as dashed lines and fade out over time. Again, they follow the order that the cascading download following, around the ring. Here node 6 is listening for heartbeats (thumps) from 9, 9 from 6, and 8 from 6.



The current system settings are hardcoded with useful debug and experimental settings. Heartbeats are sent every 5 seconds and are checked every 10 seconds. If a heartbeat is not received in 10 seconds then our "CardiacArrest" fault tolerance thread kicks in.

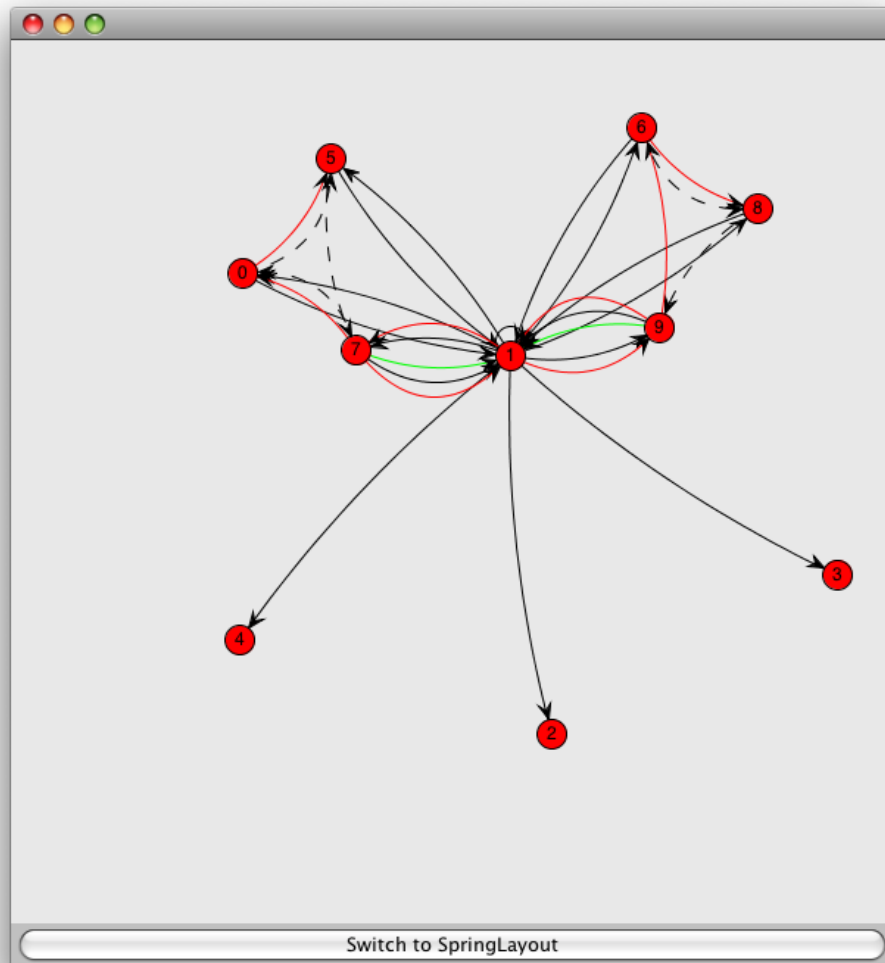
```
// Constants

/** Check Heartbeat Time */
public static final int CHECK_HEARTBEAT = 10000; /* 10 seconds */

/** Send Heartbeat Thump Time */
public static final int SEND_HEARTBEAT = 5000; /* 5 seconds */

/** Initial Send Heartbeat Thump Time */
public static final int INITIAL_SEND_HEARTBEAT = 3000; /* 3 seconds */
```

First let's issue a download command in a normal environment such as this. To issue the download command I have to provide the same filename of the file I uploaded, but not the full path. You can see in the status that I uploaded "test.txt" previously, so I'll attempt to download that file now:



Eve shows that the nodes respond perfectly. Node 1 sends out a DownloadMessage to 7 and 9 (another black line) and both respond by sending the file back to node 1 (green lines). Here is what the terminal displayed:

```
Terminal — java — 109x44
bash java java bash
>> status
Username: joe1
ID: 3996A6FA95D5A5492882F1C02D9D941416655F70
Personal File List: [raids.PersonalFileInfo@3a888c1:test.txt]
Inventory: []
>> download test.txt

Ask the following nodes for part (0):
[SNH: <0x4D4DCA..> //192.168.1.2:9009]
[SNH: <0x17BD85..> //192.168.1.2:9006]
[SNH: <0x1CC018..> //192.168.1.2:9008]

Ask the following nodes for part (1):
[SNH: <0xC9155D..> //192.168.1.2:9007]
[SNH: <0x1EFCA4..> //192.168.1.2:9000]
[SNH: <0x17982D..> //192.168.1.2:9005]

3996A6FA95D5A5492882F1C02D9D941416655F70: Expecting: 81E3063ADFBE2F58D8692249DEE19F58B006E022:0
3996A6FA95D5A5492882F1C02D9D941416655F70: Expecting: 81E3063ADFBE2F58D8692249DEE19F58B006E022:1
>> 4D4DCAC917FBF2577F4CE5DEA1A775FA2DAB94E0: received DownloadMessage
4D4DCAC917FBF2577F4CE5DEA1A775FA2DAB94E0: Sending: 81E3063ADFBE2F58D8692249DEE19F58B006E022:0 to 3996A6FA95D5
A5492882F1C02D9D941416655F70
C9155DB1E5458EFC6747EE10DD7F96483D99C74: received DownloadMessage
C9155DB1E5458EFC6747EE10DD7F96483D99C74: Sending: 81E3063ADFBE2F58D8692249DEE19F58B006E022:1 to 3996A6FA95D5
A5492882F1C02D9D941416655F70
Dumped the 4052 bytes from the buffer into the temp file >> /tmp/RAIDS-appsocketreader-20368.tmp
Dumped the 4052 bytes from the buffer into the temp file >> /tmp/RAIDS-appsocketreader-20367.tmp
Dumped the 1520 bytes from the buffer into the temp file >> /tmp/RAIDS-appsocketreader-20367.tmp
Dumped the 1520 bytes from the buffer into the temp file >> /tmp/RAIDS-appsocketreader-20368.tmp
3996A6FA95D5A5492882F1C02D9D941416655F70: Received part: 81E3063ADFBE2F58D8692249DEE19F58B006E022:0
3996A6FA95D5A5492882F1C02D9D941416655F70: Received Expected part: 81E3063ADFBE2F58D8692249DEE19F58B006E022:0
3996A6FA95D5A5492882F1C02D9D941416655F70: Received part: 81E3063ADFBE2F58D8692249DEE19F58B006E022:1
3996A6FA95D5A5492882F1C02D9D941416655F70: Received Expected part: 81E3063ADFBE2F58D8692249DEE19F58B006E022:1
3996A6FA95D5A5492882F1C02D9D941416655F70: ENOUGH PARTS SUCCESSFULLY DOWNLOADED!!!!
3996A6FA95D5A5492882F1C02D9D941416655F70: Path to chunks: /tmp/
3996A6FA95D5A5492882F1C02D9D941416655F70: Chunks to assemble: RAIDS-appsocketreader-20367.tmp
3996A6FA95D5A5492882F1C02D9D941416655F70: Chunks to assemble: RAIDS-appsocketreader-20368.tmp
3996A6FA95D5A5492882F1C02D9D941416655F70: FINISHED ASSEMBLING FILE TO '/Users/joe/test.txt'.
3996A6FA95D5A5492882F1C02D9D941416655F70: Original hash: 'a50e7ceb6c2f72549b340fd4d28ef4b7db2b07a8' new hash:
'a50e7ceb6c2f72549b340fd4d28ef4b7db2b07a8'

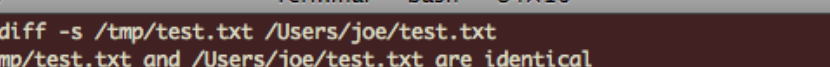
3996A6FA95D5A5492882F1C02D9D941416655F70: DOWNLOAD, REASSEMBLY AND VERIFICATION COMPLETE FOR: '/Users/joe/tes
t.txt'
```

The status shows we are still on "joe1", and again the PersonalFileList shows joe1 stores "test.txt" the file we ask for in the next step. The clean output at the top is expected output that is the result of looking up the Master List from the DHT. All the nodes in the two division groups are correctly identified. The debug below again shows the inner workings.

First and foremost joe1 is set to expect parts 0 and 1. This way when the files are sent to joe1 he will know that they are parts of a requested file. The DownloadMessages are sent out to the proper nodes, and they respond by sending the same 5572 raw file bytes and 44 PartIndicator bytes to joe1. Upon receiving the expected parts special subroutines are run instead of the normal code that would add the file to the node's inventory.

Once joe1 has received enough parts (in my case 2 because I originally chunked to 2 parts) reassembly takes place. Some of the meta-data stored in the PersonalFileList was the original hash of the file. After reassembly the reassembled file is hashed, and verified against this previously stored value. Indeed they match correctly.

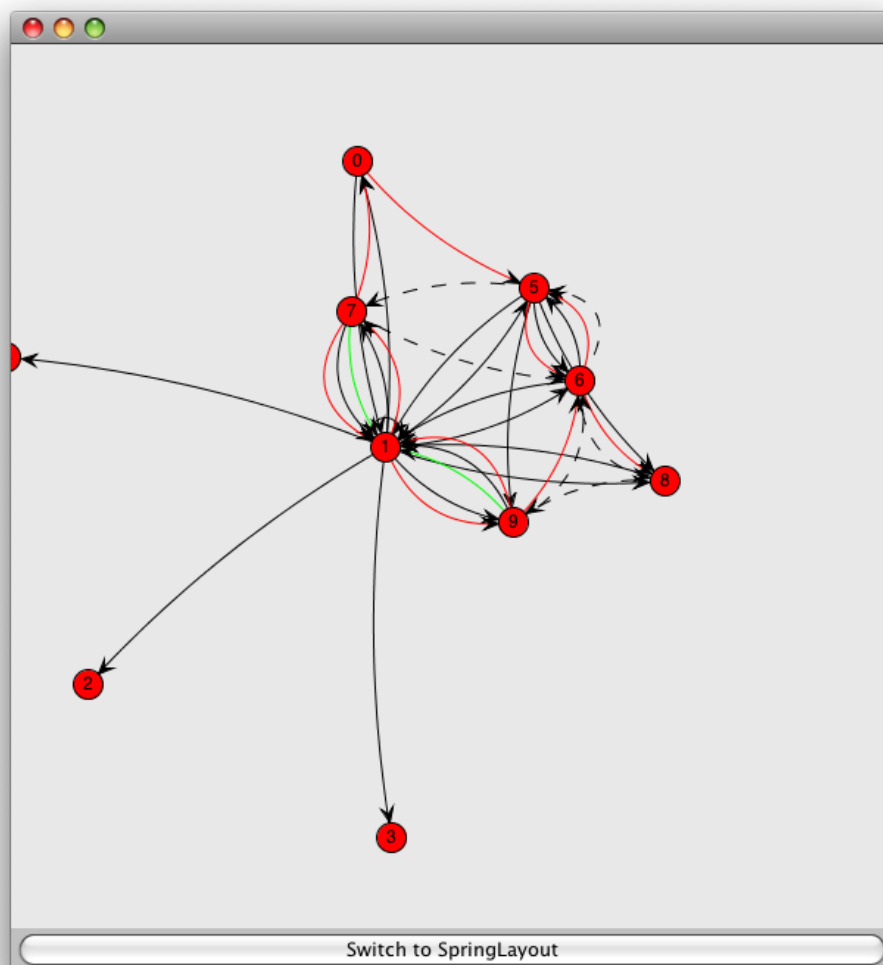
The downloaded file is stored in the user's home directory with the same name as the original file. In this case it put the reassembled file in "~/test.txt". A comparison shows they are indeed the same:



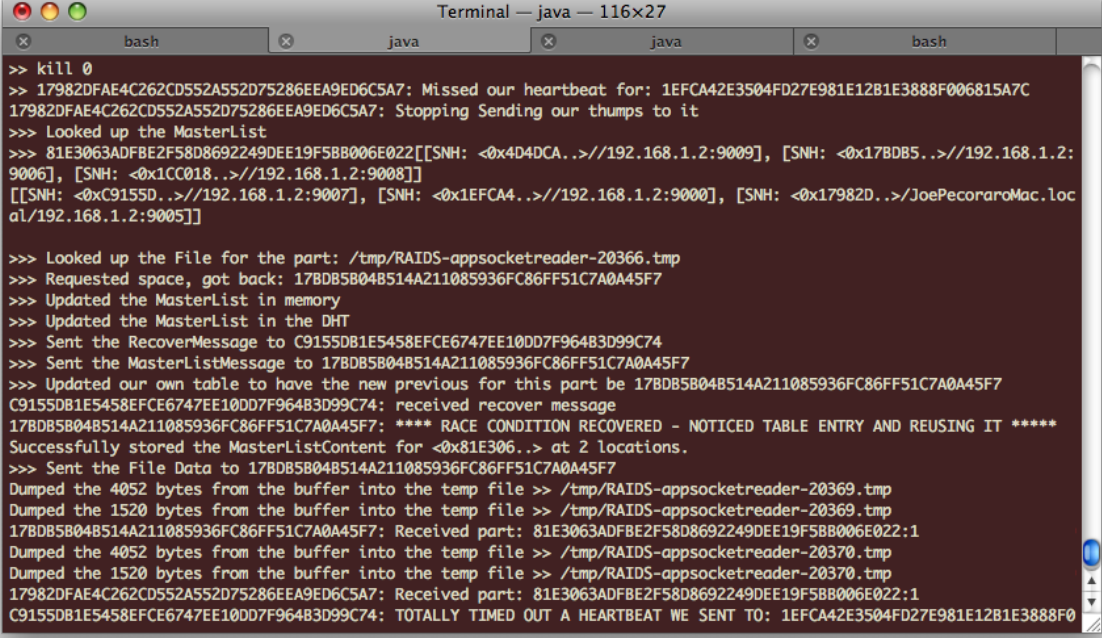
Terminal — bash — 84x10

```
joe[~]$ diff -s /tmp/test.txt /Users/joe/test.txt
Files /tmp/test.txt and /Users/joe/test.txt are identical
joe[~]$
```

Here we will invoke the Fault Tolerance procedure by killing one of the nodes in one of the rings. Lets kill node 0. When I issue the kill command you would see in the visualization that node 0 no longer sends out heartbeat messages to node 5. After 10 seconds, the timeout will set the recovery process in motion. Node 5, which is listening for node 0's heartbeats will perform the bulk of the work. Node 5 will multicast for a node with available space. Lets see what happens:



Our current algorithm only excludes nodes in the same ring and only shows the original multicast to prevent too many arrows from cluttering the visual. You can see that node 0 was replaced with node 6, who is still part of the other division group!!! The terminal output was:

A screenshot of a macOS terminal window titled "Terminal — java — 116x27". The window has four tabs: "bash", "java", "java", and "bash". The output is from a Java application, showing a series of log messages. It starts with a "kill 0" command, followed by a "Missed our heartbeat" message from node 17982DFAE4C262CD552A552D75286EEA9ED6C5A7. Then, it says "Stopping Sending our thumps to it" and "Looked up the MasterList". A large block of hex data is printed. Next, it says "Looked up the File for the part: /tmp/RAIDS-appsocketreader-20366.tmp", "Requested space, got back: 17BDB5B04B514A211085936FC86FF51C7A0A45F7", "Updated the MasterList in memory", and "Updated the MasterList in the DHT". It then sends a "RecoverMessage" and a "MasterListMessage" to node 17BDB5B04B514A211085936FC86FF51C7A0A45F7. It updates its own table and receives a "recover message" from the same node. A "RACE CONDITION RECOVERED" message is printed, followed by "Successfully stored the MasterListContent for <0x81E306..> at 2 locations." It then sends "File Data" to the same node. Finally, it dumps 4052 bytes from the buffer into a temp file and receives a "Received part" message from node 81E3063ADFB2F58D8692249DEE19F58B006E022:1. The output ends with "TOTALLY TIMED OUT A HEARTBEAT WE SENT TO: 1EFC42E3504FD27E981E12B1E3888F0".

```
>> kill 0
>> 17982DFAE4C262CD552A552D75286EEA9ED6C5A7: Missed our heartbeat for: 1EFC42E3504FD27E981E12B1E3888F006815A7C
17982DFAE4C262CD552A552D75286EEA9ED6C5A7: Stopping Sending our thumps to it
>>> Looked up the MasterList
>>> 81E3063ADFB2F58D8692249DEE19F58B006E022[[SNH: <0x4D4DCA..> //192.168.1.2:9009], [SNH: <0x17BDB5..> //192.168.1.2:
9006], [SNH: <0x1CC018..> //192.168.1.2:9008]]
[[SNH: <0xC9155D..> //192.168.1.2:9007], [SNH: <0x1EFC4A..> //192.168.1.2:9000], [SNH: <0x17982D..> //JoePecoraroMac.loc
al/192.168.1.2:9005]]

>>> Looked up the File for the part: /tmp/RAIDS-appsocketreader-20366.tmp
>>> Requested space, got back: 17BDB5B04B514A211085936FC86FF51C7A0A45F7
>>> Updated the MasterList in memory
>>> Updated the MasterList in the DHT
>>> Sent the RecoverMessage to C9155DB1E5458EFCE6747EE10DD7F96483D99C74
>>> Sent the MasterListMessage to 17BDB5B04B514A211085936FC86FF51C7A0A45F7
>>> Updated our own table to have the new previous for this part be 17BDB5B04B514A211085936FC86FF51C7A0A45F7
C9155DB1E5458EFCE6747EE10DD7F96483D99C74: received recover message
17BDB5B04B514A211085936FC86FF51C7A0A45F7: **** RACE CONDITION RECOVERED - NOTICED TABLE ENTRY AND REUSING IT ****
Successfully stored the MasterListContent for <0x81E306..> at 2 locations.
>>> Sent the File Data to 17BDB5B04B514A211085936FC86FF51C7A0A45F7
Dumped the 4052 bytes from the buffer into the temp file >> /tmp/RAIDS-appsocketreader-20369.tmp
Dumped the 1520 bytes from the buffer into the temp file >> /tmp/RAIDS-appsocketreader-20369.tmp
17BDB5B04B514A211085936FC86FF51C7A0A45F7: Received part: 81E3063ADFB2F58D8692249DEE19F58B006E022:1
Dumped the 4052 bytes from the buffer into the temp file >> /tmp/RAIDS-appsocketreader-20370.tmp
Dumped the 1520 bytes from the buffer into the temp file >> /tmp/RAIDS-appsocketreader-20370.tmp
17982DFAE4C262CD552A552D75286EEA9ED6C5A7: Received part: 81E3063ADFB2F58D8692249DEE19F58B006E022:1
C9155DB1E5458EFCE6747EE10DD7F96483D99C74: TOTALLY TIMED OUT A HEARTBEAT WE SENT TO: 1EFC42E3504FD27E981E12B1E3888F0
```

Here the output is nearly all cryptic because all of the details happen behind the scenes. Briefly, from top to bottom you can see that 17982... or "joe5" reported missing the heartbeat from 1efca... or "joe1." At this point "joe5" assumes the medic role. Step 1 is to look up the Master List. The protocol then proceeds rather linearly. The medic finds the file on his own machine, multicasted and got back a reply from 17bdb... or "joe6." The DHT record is updated, recovery messages are sent out as well as the file chunk.

Two things are noticeable here. First is that the file was accidentally cascaded. This was a bug in our application that we were aware. The second is the potential race condition recovery message on "joe5." This is either another bug or a real notification of recovering from a race condition we are aware of. Either way, both are harmless and lead gracefully to a good state.

Now node 6 is part of both division groups, maintaining two separate heartbeat listeners and senders, and storing two parts. This gives us an opportunity to check the inventory of that node to see if this is indeed the case:

```
Terminal — java — 103x26
bash java java bash
Username: joe5
ID: 17982DFAE4C262CD552A552D75286EEA9ED6C5A7
Personal File List: []
Inventory: [81E3063ADFB2F58D8692249DEE19F5BB006E022:1]

Username: joe6
ID: 178DB5B04B514A211085936FC86FF51C7A0A45F7
Personal File List: []
Inventory: [81E3063ADFB2F58D8692249DEE19F5BB006E022:0, 81E3063ADFB2F58D8692249DEE19F5BB006E022:1]

Username: joe7
ID: C9155DB1E5458EFCE6747EE10DD7F96483D99C74
Personal File List: []
Inventory: [81E3063ADFB2F58D8692249DEE19F5BB006E022:1]

Username: joe8
ID: 1CC018AF7DF4E9CC969F43F57B0713A264DDA607
Personal File List: []
Inventory: [81E3063ADFB2F58D8692249DEE19F5BB006E022:0]

Username: joe9
ID: 4D4DCAC917F8F2577F4CE5DEA1A775FA2DAB94E0
Personal File List: []
Inventory: [81E3063ADFB2F58D8692249DEE19F5BB006E022:0]

>> 
```

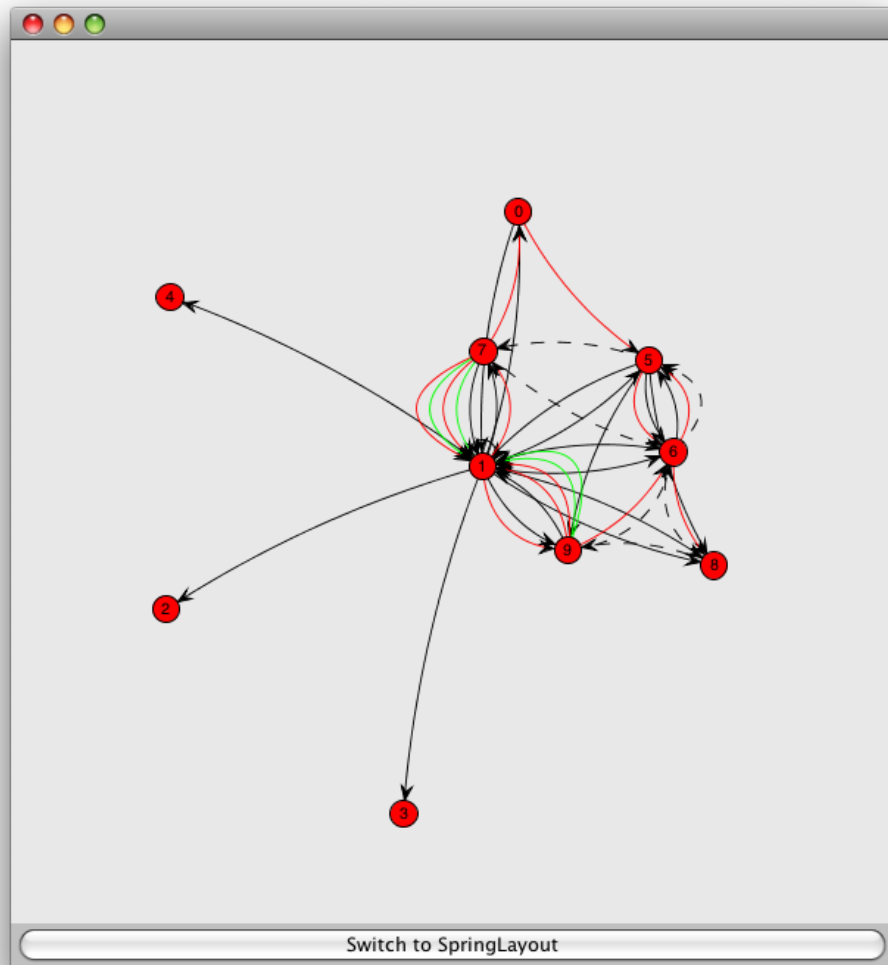
Yes, this shows joe6's inventory is storing both part 0 and part 1 of the same file. This is fine due to the replication of the file across enough nodes. If node 6 were to die there would still be more than enough replicas of the chunk on the network.

Next I decide to join the ring from what looks like a different computer. I'll start a new instance with username "joe1" connecting to the existing Pastry network formed by the original 10 nodes. It correctly shows that I have file "test.txt" in my Personal Inventory:

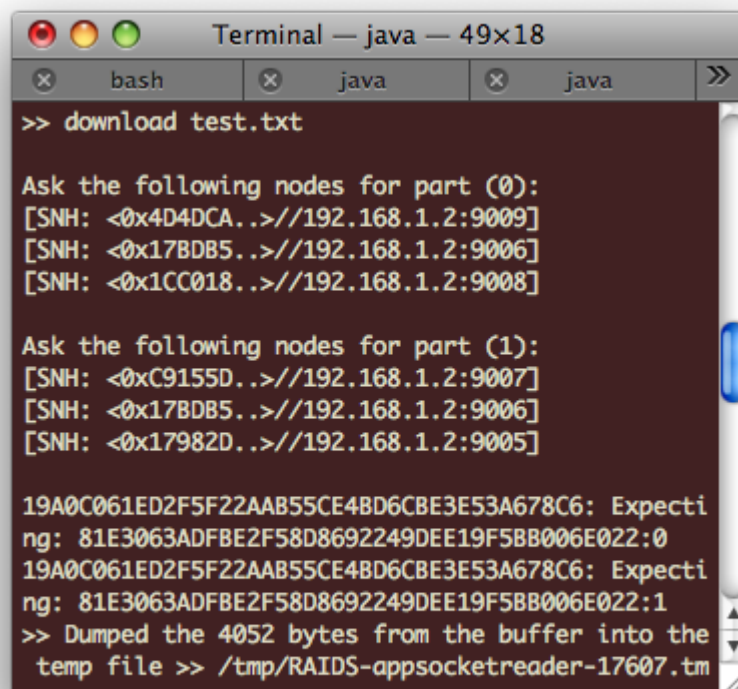
```
Terminal — java — 96x9
bash java java bash java
joe[~]$ java raids.Client 9050 localhost 9009 joe1 1 localhost 9999
New Client
Finished creating new node PastryNode[SNH: <0x19A0C0..>/JoePecoraroMac.local/192.168.1.2:9050]
>> status
Username: joe1
ID: 19A0C061ED2F5F22AAB55CE4BD6CBE3E53A678C6
Personal File List: [raids.PersonalFileInfo@1822b7f8:test.txt]
Inventory: []

>> 
```

Although the two "joe1" users are technically on different nodes we programmed Eve to join them both as the same logical vertex. So when I issue the download command on the new instance, it shows the updated DHT values written by the medic earlier when node 0 was killed, and the downloads happen normally:



And the terminal showing the DHT values:

A screenshot of a macOS Terminal window titled "Terminal — java — 49x18". The window has three tabs: "bash", "java", and "java". The terminal output shows a command prompt ">> download test.txt" followed by two requests for data parts from nodes. The first request is for part (0) and the second for part (1). Both requests are received from three nodes with different SNH values. The output then shows two identical lines of hexadecimal data, each followed by a confirmation message. Finally, a message indicates that 4052 bytes were dumped from the buffer into a temporary file.

```
>> download test.txt

Ask the following nodes for part (0):
[SNH: <0x4D4DCA..>//192.168.1.2:9009]
[SNH: <0x17BDB5..>//192.168.1.2:9006]
[SNH: <0x1CC018..>//192.168.1.2:9008]

Ask the following nodes for part (1):
[SNH: <0xC9155D..>//192.168.1.2:9007]
[SNH: <0x17BDB5..>//192.168.1.2:9006]
[SNH: <0x17982D..>//192.168.1.2:9005]

19A0C061ED2F5F22AAB55CE4BD6CBE3E53A678C6: Expecti
ng: 81E3063ADFBF2F58D8692249DEE19F5BB006E022:0
19A0C061ED2F5F22AAB55CE4BD6CBE3E53A678C6: Expecti
ng: 81E3063ADFBF2F58D8692249DEE19F5BB006E022:1
>> Dumped the 4052 bytes from the buffer into the
temp file >> /tmp/RAIDS-appsocketreader-17607.tm
```

This concludes the walkthrough.

Results

RAIDS has been successfully tested in two environments. The first environment consists of a single computer running multiple virtual machines. Our second set of tests occurred on the RLES environment. Using the RLES we were able to connect multiple machines together to simulate a network of computers connected in our virtual network.

We successfully built:

- A RAID striping algorithm that uses parity information to recover lost data.
- A file distribution and maintenance application on top of the FreePastry peer-to-peer substrate.
- A command line Client program that combines the previously mentioned components with FreePastry's PAST and SCRIBE applications resulting in a reliable per-file distributed backup system.

Future Work

We are certainly pleased with our results. However, if this project were to continue we have identified a number of areas which can be improved, necessary features that should be implemented, and potential weaknesses that should be resolved.

- Security - Both for the messages and file transfers, as well as for the integrity of the peer-to-peer network when confronted with malicious nodes. We envision public key encryption for file data that would guarantee that only the user who uploads a file would be able to request and reassemble the file.

- Scalability - As the professor pointed out, one weakness that our system currently has is that any arbitrary set of nodes may form a heartbeat ring. This has potential scalability issues that could be resolved with a better design for rings. A likely solution would be restrict node participation to only a few rings, or clusters, and uploaded files would be assigned to such a cluster rather than any arbitrary nodes to form their own ring.
- Session Persistence - Our application right now transfers files and stores those files but it makes no effort to maintain session data between runs. That was beneficial for a debugging and developing standpoint, but its unrealistic for real usage. Configuration and per-user information should be stored persistently for real usage.
- Modularized Backup Algorithm - Many of the algorithms in our application could be tweaked or substituted with another. One in particular is our RAID striping algorithm. Our application right now is moderately coupled to this algorithm but we foresee the potential to allow the application to support any RAID-like storage algorithm. This idea of swappable algorithms would allow the application to work effectively in many different environments. Such as ones with latency issues or ones with larger storage capacities.
- Performance - As always, the performance and memory footprint of the application can be improved. Some areas in particular would be the file distribution. We chose to "cascade" files because the approach makes sense. However, there are other algorithms, such as those used in BitTorrent that would allow for faster distribution of files to nodes. Since latency is a major factor in performance this would be a worthwhile improvement.

Lessons Learned

Our group learned a tremendous amount during the development process of this application. We learned a lot about peer-to-peer systems in general, not just specific to FreePastry. FreePastry implements a standard peer-to-peer application interface that exists across many peer-to-peer implementations. This means that much of the experience we gained with FreePastry is applicable to other peer-to-peer systems.

Besides the peer-to-peer specific interface we gained experience with message passing and the programming paradigms associated with it. We thought a lot about potential race conditions and synchronization problems that could be a result of the asynchronous message passing used for general peer-to-peer communication.

We also went through a number of iterations during the design process in order to end up with a system that was fault-tolerant and could reliably upload and download files. An example of this was that our original plan did not include any erasure coding system, we wanted to ensure redundancy by replication on the nodes. Later we decided that if we could split and recover from errors with an erasure system it would increase our fault tolerance and the overall performance. However, we couldn't get the more advanced erasure coding system to work so we settled on a combination of erasure and replication for the storage of files on our system. In the end it was the hours of design work that allowed the project to succeed. Every time we ran into a problem with something we already had fall-back solutions to rely upon.

In addition to the technical aspects we learned a great deal about collaboration software and open source research projects. We had some problems with code conflicts in our group because of the pace at which the project had to be developed. Everyone had to add items to almost every file during the course of the project. So, in order to resolve the conflicts and prevent future ones, constant communication and meetings were necessary. We also took away a lot from our experience with FreePastry. While it definitely worked and made it possible for the project to succeed, FreePastry is still a work in progress and has some issues that were outside our control. We ran into a large

number of issues with the API and tutorials on the site. The tutorials use deprecated methods--many of which have no clear replacements. There were also bugs in the actual implementation of FreePastry that required us to work around or fix them. All of these issues were a result of using a research project rather than a production piece of software.

We used Eve as a message logger for the debugging of our project. This became a problem if we trusted Eve too much, and Eve contained an error, then we would incorrectly assume the problem was with the RAIDS protocol. There is clearly a price to pay for developing a secondary application to help visualize or debug. However, in the end Eve was a very helpful debugging and visualization tool that effectively and accurately shows how our application work. It is much more user friendly, and therefore easier to understand, than text output on the terminal.