# ColonyFS: An Anonymous, Secure and Dependable File Store for Peer-to-Peer Networks

Andrew Lyons - al77@kent.ac.uk, Andrew March - ajm45@kent.ac.uk,
Chris Nolan - cdn3@kent.ac.uk, Martin Ellis - mce4@kent.ac.uk

***Abstract*—Distributed file stores are used when data accessibility and redundancy are required. However there are currently few solutions available for distributed file stores that emphasise security and anonymity of data. To address this issue, we designed and implemented a peer-to-peer file store providing these features using Fragmentation Redundancy Scattering and a naturally inspired distribution algorithm.**

## I. Introduction

TRADITIONALLY secure file stores have been based around centralised server systems. These systems suffer from being expensive (requiring specialised hardware) and require trusting of system administrators to control access and perform backup functions[1]. For end users, the alternative is to establish their own storage system, which requires expense and expertise. In contrast, peer-to-peer systems by their nature allow independence from third parties, and are able to pool the resources of many machines. In recent years research shows that an increasingly large proportion of internet traffic has been dedicated to peer-to-peer networks [2], indicating that users are becoming more accepting of the technology. We have attempted to leverage the greater acceptance and clear advantages offered by peer-to-peer networks to design and implement a distributed file storage system, with an emphasis on security, anonymity and dependability. In order to achieve these qualities, we used a technique called fragmentation redundancy scattering and an optimisation algorithm inspired by the movement of ants.

In this paper we detail the design and implementation of this system, including the techniques used and some of the technical challenges involved. We also discuss some conclusions and potential further work.

## II. Background

Our project is primarily based on the ideas presented in the papers "Dependable and Secure Distributed Storage System for Ad Hoc Networks"[3] and "Secure Agents in a Distributed Storage System"[4], which investigated in detail a variety of algorithms for the production of a distributed, dependable storage system. While these papers were mainly focused on the theoretical issues surrounding these systems and were therefore limited to simulations, our system provides a practical peer-to-peer implementation over heterogeneous networks such as the Internet. Both of these works describe how fragmentation redundancy scattering and the ant inspired algorithm can be employed in such a system, and we will describe later in this paper how we applied these ideas in our implementation.

### A. Fragmentation Redundancy Scattering

Fragmentation redundancy scattering (FRS) is a method of distributing data around a network, by splitting it into encrypted fragments and scattering these across many machines. Individual fragments contain no significant data, therefore they cannot be used to recover any of the original data nor identify the original owner. Each fragment is stored in multiple locations to provide protection against failures and data loss. The redundancy arises from the ability to restore data from a number of sources. For example the loss of a subset of the machines on the network does not prevent the original data from being recovered, as its component parts may be recovered from multiple locations.

### B. Ant Inspired Algorithm

To achieve the aims of anonymity and security we have used an ant-inspired algorithm to distribute and retrieve data fragments. This approach allows us to hide the source, destination and route of the data. The three component parts of the algorithm are insertion, replication and retrieval. Insertion is used to add data to the network, with retrieval used to get the data out. Replication implements the redundancy component of FRS by maintaining multiple copies of fragments, and ensuring they are widely distributed away from the source.

## III. Related Works

### A. Freenet

The Freenet project has many similar aims to what we wanted to achieve with ColonyFS. It aims to provide anonymity of data for both insertion and retrieval (it is infeasible to discover the true origin or destination of a file passing through the network[5]), be resistant to third party attempts to change or deny access to data, and have a fully decentralised network. It differs however in that data is not guaranteed to survive on the network. This is because in the Freenode project replication is dependant on access, that is data is only replicated when it is requested. This means that data that is frequently requested will have the highest number of replicas whilst data that is no longer accessed will eventually disappear from the network. In contrast, the replication process in ColonyFS is automated and all fragments are replicated equally.

### B. Tor

Tor[6] (The Onion Router) is an example of a distributed routing algorithm we researched whilst designing ColonyFS. Tor's algorithm uses encrypted messages containing the complete route for a packet, this is implemented in such a way that each each node in the path knows its predecessor and successor, but no other nodes in the route[7]. We investigated the feasibility of designing ColonyFS's routing algorithm using similar ideas, requiring each message to carry its complete route through the network. However unlike the Tor network in which routes are calculated in advance, our routes are dynamically calculated and in some cases these routes may be very long, resulting in messages of a large size. We also knew that our ant inspired algorithm was extremely network intensive, so decided to use a distributed routing method where the routing node made independent routing decisions.

## IV. Aims

To structure the development of our system we divided our aims into three groups; core, desirable and extended.

Core
: These aims are key to the system, and we should achieve these as a minimum.

Desirable
: These are features that would be good to have in the system, however aren't key to its success. We will implement these if we find we have time.

Extended
: These are possible further extensions to the system, which would provide some added functionality but aren't important for what we're doing.

### A. Core

- Implement a distributed file store with the ability for users to insert and retrieve files.
- Implement FRS to provide security and redundancy.
- Implement an ant inspired algorithm to distribute fragments.
- Provide integrity assurance at the fragment and file level to warn of compromised data.
- Develop a well featured interface to manage the user's interaction with the system.

### B. Desirable

- Automated node discovery analogous to BitTorrent's tracker system in a node centric as opposed to file centric manner.
- Production of a structure document to provide a persistent data structure for a user's virtual file store.
- Implement file deletion and lifetime to prevent the stagnation of the system.
- Introduce authentication measures to permit users to log in to access their virtual file store.
- Develop advanced interface features to improve integration with native operating system.
- Produce alternative interfaces to work with a variety of platforms.

### C. Extended

- Offer user selectable encryption options.
- Investigate further optimisation techniques that could be employed to improve performance.
- Improve the failure handling of the system to ensure robustness.

## V. Implemenataion

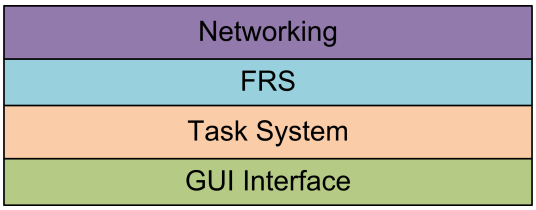
The implementation of the project can be divided into two key areas; the client application and the user GUI. The GUI is a lightweight display component that accepts user input and then sends requests to the client application to perform the necessary processing, such as the actual insertion and retrieval of files. The client application then sends the results of the processing back to the GUI for feedback to the user. We separated these components as they have different requirements, and it allowed us to implement them in languages best suited to their functionality. It also gave us the flexibility to easily connect different front-end GUIs without having to provide different distributions of the same core code base.

## VI. Passport File

One problem we faced was how to represent the user's file structure internally, and how to share this between the client and GUI components. We came up with the idea of providing a "passport" to the system; a file belonging to the user which provides all the information they need to be able to retrieve their files. This "passport file" contains a summary of all files the user has put on the system and the fragments that make up the file with their hashes. We implemented the passport file using XML, as it allows easy description of a recursive schema that defines the structure of the file system, allowing users to have a file structure they are familiar with. The passport file also allows support for file "revisions", where one file has multiple revisions listed, and each revision has its own list of fragments. Due to the way our system works, old versions of files are never lost, so we can request them by their revision number and retrieve historical data

## VII. Modular Design

One of the key design consideration for this project was making the system as modular as possible. The reasons for this were two fold, firstly this makes any future work reusing our project as simple and possible, and this enabled us to work on independent modules of the code base while we finalised the design of other parts of the system.




The client application can be coarsely split into four main modules. The networking module handles the formation of the network as well as interaction with neighbouring nodes and

controlling fragment replication. The FRS module implements the tools required to perform the tasks of the FRS algorithm. The Task Management system is used to control and execute the instructions received from the GUI via the GUI Interface module which handles communication with the GUI using TCP.

To achieve this modularity we employed a number of techniques including the use of an inversion of control container (The Spring Framework) and the Enterprise Integration Pattern (Spring Integrations). Spring Integrations allows us to communicate between modules using message passing, simplifying the interface between them and helping to solve concurrency issues.

## VIII. Networking Module

Ideally for our system we would build our own peer-to-peer networking library, however due to the limited time we had to complete our work we decided to use an existing middleware solution. Initially we could only find one option, JXTA, which is a set of open protocols that enable any connected devices on the network to communicate and collaborate in a P2P manner[8]. JXTA was originally developed by SUN, but has since been turned into an open source, community driven project.

Significant attempts were made to configure, setup and run sample JXTA code, so that a feeling could be gained for its capabilities and complexities. However, a chronic lack of documentation, waning community, and sheer complexity of the project made implementing any solution, even simple, overly complex for our needs. While SUN have visions of making JXTA the de-facto standard for Java P2P, they and the community need to give thought to improving the documentation provided for newcomers.

Fortunately, we found another option in the form of the Pastry P2P system. Pastry is a "generic, scalable and efficient substrate for peer-to-peer applications"[9]. It provides a ring overlay network, which is decentralized, self-organizing and fault tolerant. Our ideal solution for anonymity and routing purposes requires a more traditional mesh-like network layout, therefore Pastry was not a perfect solution for us, however we were able to use it as a foundation for providing our networking layer.

Pastry provides the capability to send messages directly between nodes on the network, therefore to build a mesh-like overlay on top of pastry each node needs to know about a limited subset of other participants (which it considers "connections"). To build these connection lists in the nodes we use Scribe, an application-level multicast system that is built of top of Pastry [3]. Our nodes all subscribe to a single multicast group upon connecting to the network, and then proceed to broadcast their pastry node handle at a regular interval. Other nodes will be receiving these messages constantly, and will randomly choose some of them (up to a defined limit) to be "connections". The diagram in Figure 1 shows a typical layout we tried to achieve; the black circles represent nodes, the blue lines represent free pastry connections, and the black lines represent connections in our overlay.

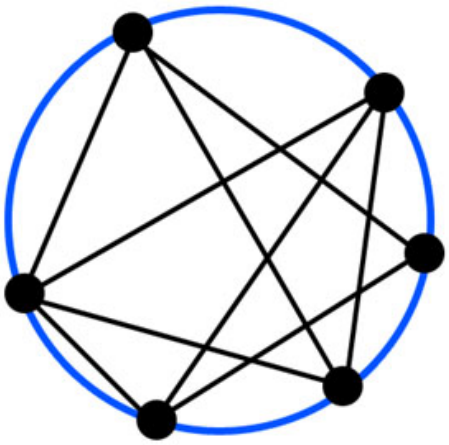


Figure 1. Our Free Pastry Overlay

In our initial implementation these connections were one way forwarding connections. They would be used to make outgoing routing decisions, but if a message needed to be returned a node would use an internal record of the sending node rather than its list of connections. While this worked for small networks, on larger networks we found that nodes or groups of nodes could become isolated, and that our routing quickly failed due to the limited number of available paths. To improve this we implemented bidirectional connections. When a node received a multicast message containing a node address that it would like to connect to it added the node to its list, and then sent a message to the destination node asking it to open a return connection. This greatly improved the operation of the network, giving us true mesh topologies and making the routing algorithm significantly more effective.

Node discovery is an issue in P2P networks. Pastry itself provides no centralized name-service for a potential node to use to lookup an initial node to connect to (unlike say Bittorrent whose trackers provide this functionality). Therefore we built a simple PHP script that upon querying provides a list of possible hosts for a node to "bootstrap" from. When a node has connected it has a chance of calling this script again to add its own data to the list. This PHP script was hosted on one of the group's remote servers to simulate a scenario as realistic as possible.

### A. Message Types

To propagate and retrieve fragments from the network we use three types of messages; replication, outgoing retrieval and return retrieval ants.

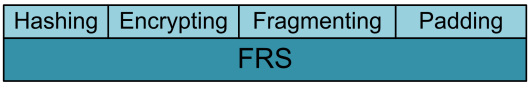
Replication Ant

The Replication Ant performs two tasks; as the name suggests, it is used for replicating fragments around the network but it is also used for the initial insertion of fragments. This is possible (and desirable) because insertion is essentially the same as replication, and makes monitoring insertions difficult because insertion traffic is interleaved with regular replication traffic.

Outgoing Retrieval Ant
: The Outgoing Retrieval Ant preforms a depth first search of the network looking for specific fragments using their hash for lookups.

Returning Retrieval Ant
: Once an Outgoing Retrieval Ant has found the fragment it was sent to retrieve it is transformed into a Returning Retrieval Ant which carries the fragment back to the originating node by following the route it took on its outgoing search.

## IX. The FRS Module




The FRS tools module provides facilities to accomplish each task needed in preparing files and reconstructing files.

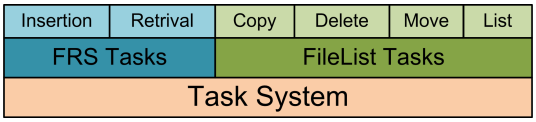
Padding
: To ensure that all fragments are the same length (this prevents attacks on the last fragment in a file) we pad the file to a multiple of the fragment size.

Encryption
: We use AES to encrypt the padded file, to prevent inspection of distributed fragments.

Fragmenting
: Files are split into fragments of equal length for distribution on the network.

Hashing
: We use SHA-digest to obtain a hash for each fragment. This is used as the fragment identifier which is recorded in the passport file, as well as being used for integrity checking.

## X. The Task System Module




The Task System is used to manage requests (tasks) from the GUI. It is composed of two parts, the task manager and a set of tasks. The task manager is responsible for initialising tasks using the options sent from the GUI and running each task in its own thread. We run tasks concurrently for two reasons, firstly because this is how users expect to be able to interact with systems like ours, but also because running tasks in parallel interleaves requests to the network, helping obscure what data relates to what tasks. As well as the task manager there are the tasks it runs, which are logically split into two groups.

FRS functions
: The FRS functions are the most complex of the tasks. They control each stage of the FRS algorithm and report their statuses back to the GUI.

File List Tasks
: These tasks do not interact with the network but just make logic changes to the passport file.

## XI. The GUI Interface Module

To maintain the cross platform nature of the system without compromising the user experience we opted to develop the user interfaces in languages best suited to each platform. This required us to pass messages from the client application written in Java to interfaces written in variety of languages. To achieve this functionality we investigated the use of named pipes, however the implementation of this technology varies between platforms requiring us to develop different implementations dependant on the deployment platform. We also investigated messaging over TCP; this provided us with the same functionality as named pipes but in a platform and language independent way. One assumption we made with this implementation was that the user's local machine could be trusted.

### A. Messaging Protocol

For the GUI and client application to communicate effectively we needed to define a messaging protocol they could both adhere to. This needed to be both concise enough to remove any ambiguity of intent yet flexible enough to allow further extensions. We decided to implement the messaging by passing XML messages between the systems, as XML is easily extendable and provides a universally recognised format that would be suitable for any later environments we developed interfaces for.

We came up with an XML schema that allowed us to describe all the tasks the GUI could conceivably want to process (See corpus material). Every message contains an "Action" element, with "type" and "id" parameters. The "id" parameter is used to link multiple messages relating to the same task together, so when the GUI receives an "insertion complete" response from the client, it knows which file has finished inserting. The "type" parameter dictates what action the GUI wants to carry out. So, for example, to insert a file, the GUI would specify an "insert" action type, or to retrieve a file, the GUI would specify a "retrieve" action type. Further lines in the XML then describe the specific details required for that action. To use the file insertion example again, within the "Action" element is a "File" element, with various parameters detailing the file to be inserted, its location in the passport file, and the files expiry date. Different action types have their own required elements defined. By using this method of Actions and child elements, we can construct complex yet flexible messages that are simple to process on both ends.

For the client application's responses to the GUI, we use a simple "Response" element, containing a "Status" element and a "Content" element. This structure allows us to model any conceivable response, as the "Content" element is additional data and could contain anything, provided the GUI knows how to handle it. The "Status" element provides a simple OK/ERROR status flag, which could be extended if there is a need in the future. The "Content" element's content is therefore dependent on context; if the status is OK, then the content could contain nothing, or some details about what the client is currently doing. We make use of this for the progress bars in the GUI; for most tasks, the client is constantly sending

progress updates telling the GUI the percentage completeness. If the status is ERROR, then the "Content" element contains error details for the GUI to feed back to the user.

## XII. User Interface Implementations

### A. Windows GUI

We targeted Windows with our first interface implementation, in producing this we considered developing either a full feature GUI or providing the same functionality through an extension to Explorer. A full GUI would provide us a greater range of control over the design allowing us to present a richer user interface, however would separate the application from the file system requiring users to make a logical leap. An Explorer extension would avoid this requirement as the application would integrate with the native file system but would be limited in regards to providing feedback as to the program's actions. A decision was reached to implement a fully featured GUI using C#, this allowed for the program to take on the look and feel of the native operating system while providing the user with a rich interface which we felt would better suit the complexity of the system.

*1) GUI Design:* The interface is divided into two tabbed panes; the first provides a tree structure of the user's virtual file store and the second displays the active and completed tasks. The file view has been designed to provide an experience that is as similar to Explorer as possible to maintain user expectations of a file store. The second pane provides the user feedback from the application regarding task status, we felt that this was important as some tasks may not be instantaneously completed.

On request from the GUI the user's file list is transferred from the client, this is then parsed to populate the tree structure of the first pane. The nature of the file list document makes it straightforward to provide a true representation of the structure of a file system despite the underlying lack of structure in the system itself. Double clicking any item in this tree will present a dialog to retrieve that file; this dialog allows the user to specify a local storage location and the revision of the file to retrieve. When any action such as this is triggered the GUI adds the item to its work queue which is an internal record of the actions the user has performed in that session.

User driven events are displayed in the work queue tab; allowing the user to review the progress of any active requests. The work queue is made up a series of custom built controls; by doing this we were able to achieve complete control over the appearance of the object. The first control provides a tabular structure designed to nest another set of further controls which each display the status of an individual task. These controls consists of an icon indicating the type of operation, the virtual file name, a state descriptor, a progress bar and a context menu. This provides an easy to read overview of each operation and its current state. The context menu allows us to provide an action list specific to each operation without cluttering the main display. The context menu is triggered by right clicking on an active request. Through this menu the user can view more information, cancel an active operation or open a completed retrieval. The more information option presents a dialog where we are able to provide detailed data about a particular request including the time the operation began and finished, the local location of the file and a more descriptive message as to the state of the transfer. In the event of an error this allows the persistence of the error message providing insight into what caused the failure. When a retrieval is successfully completed an "open with…" option is added to the context menu, this allows the user to launch the standard Windows dialog to open files, if a default has been set for a file type this will automatically open the file in that program.

Through close interaction with standard Windows functionality we have been able to provide a consistent user experience without sacrificing the level of feedback we are able to provide to the user.

### B. Java Command Line

A command line interface has been developed in Java, this is less featureful implementation than that found in the Windows GUI. However it provides the majority of basic functions required to interact with the system. This was produced primarily to facilitate testing on non-Windows systems such as OS X and the University's Pi cluster which runs Linux.

## XIII. Testing

### A. System Testing

Testing is an important part of developing a system as complex as ours. In order to ensure our system operates as specifed, we have built unit tests around some of the functionality we felt was suitable. Our unit tests were built with the Junit[10] testing framework, and we used Emma[11] to detect code segments lacking test cases. We also frequently ran through a set of tasks to test the end to end functionality of the system. These tests included bringing up a network, file insertion and file retrieval. We also developed a separate monitoring application to allow us to examine the operation of a running system.

### B. Universities Pi Cluster.

We made use of the of the cluster which allowed us to easily bring up 30+ nodes, this allowed us to spot topology/implementations errors that did not arise on smaller networks.

### C. Monitoring Tool

Due to the complex nature of the system, it became very difficult to trace what was happening during testing. You could insert a file and immediately lose track of its fragments as they bounced around the network, and log messages were little help due to the sheer amount of information they presented (it was impossible to follow the path of a particular fragment in the multi-threaded, multi-device environment). We encountered issues early on with disappearing ants, long retrieval times and disconnected networks, and had no easy way to debug the problems and find out where they were stemming from. To remedy this, we developed a monitoring system with two components:

- An aggregation application to connect to all nodes on the network and record their data into a database.
- A monitoring GUI to read the data from the database and display it in a simple and intuitive manner.

*1) Aggregator:* The aggregator application is a Java application that was needed to retrieve data from all nodes on the network and record it into a database.

We initially decided to approach this problem by building our data model and database structure, as this would give us a good foundation on which to develop the monitoring further. The database structure ended up being quite complex (See corpus matirial), especially in terms of relationships between entities, and it quickly became apparent at this stage that raw SQL queries to insert and retrieve data were going to be very unwieldy.

Because of this we decided to investigate object-relational mapping software for Java, which we hoped would simplify development. Two solutions were considered, Hibernate and Apache Cayenne. Hibernate is "a powerful, high performance object/relational persistence and query service"[12], and forms part of the JBoss Enterprise Middleware System. Apache Cayenne is "an open source persistence framework....providing object-relational mapping and remoting services"[13].

Some previous experience with Hibernate allowed us to move forward very quickly, however we soon became hampered by the amount of setup required. The setup issues included complex XML definitions and no way of synchronising the code and the database models of the data. A quick investigation of Cayenne showed us that we could utilise a tool they provide to build our database and generate all the configuration required. As we had become aware by this point that some changes to the data model would be needed, we considered that the tighter integration between the code and database, via the Cayenne modelling tool, would be more beneficial for us.

Initially the system was designed so that the aggregator would make outgoing connections to the nodes it was to monitor. This worked reasonably well, but required the aggregator to store a list of all nodes to be monitored. To simplify setup we decided to switch the connection topology around; instead of having to maintain a list of running nodes with the aggregator, each node would be configured with the network location of the aggregator. This meant that the aggregator would scale better, accepting any number of incoming connections from remote nodes, and be easier to administer.

Collecting and storing the large amounts of data we required to build up an accurate picture of a running network posed significant challenges. Not only was the sheer quantity of data an issue, but also the timeliness in which that data had to be stored; this was important if we were to be able to accurately track message paths through the network.

To resolve the timeliness issue, each node was represented as its own processing thread within the aggregator. Each message would be time-stamped as soon as it arrived and then queued up to be processed. In practice we found that messages arriving out of order was a problem, as required database relationships could not be resolved in some cases. We therefore updated the aggregator so that it could delay handling of a message if it could not be processed initially. In addition, significant amounts of caching were implemented to improve processing time.

Through successive iterations of development, the aggregator application can now handle a network of 30 nodes in near real-time. We believe it can scale higher, however we lack the facilities to test this.

*2) Monitoring GUI:* The monitoring GUI was designed to give developers a visual front end to the data in the database, to easily follow the activity on the system. It was designed in Java Swing due to the quick turnaround with which we could produce a working system, and the availability of graphing libraries for Java. A web-based interface was considered, and would be preferable, however no graphing library (including automated graph layout) could be found that was suitable for us, and it was decided that a Java GUI would have a faster turn around for what is effectively a development tool, with no consumer facing aspect.

A key part of the GUI is a network map, which shows a graph of all nodes connected to the system and the connections between them. This graph display was produced using a customised version of an open source library named TouchGraph. TouchGraph was chosen as it already provided a lot of functionality we required, yet was flexible enough to handle various modifications we needed to make it more suitable for our uses. It is also built as a Swing component which made it easy to use in our application.

As well as the graph display, there are also lists of all files being stored on the network and the fragments that make up those files. The lists are fully interactive and communicate with the graph display, allowing users to highlight particular nodes on the network and view their files and fragments they are currently storing, highlight particular files and view where they originated, or highlight particular fragments and view where they are currently stored. The user can also set the time the system uses to any past moment, in order to view the historical system layout.

The GUI also has a second key part, the 'ant tracking pane', which allows the tracking of ants' routes through the system. Design for this part of the system was challenging, due to the large volume of data it was expected to display; it needed to show a list of every ant message that was created on the network, and then a full list of every node that ant visited and what action was performed there. We settled on using a tree-view design, which could be continually expanded to provide further details on ants. This would stop the interface from becoming too cluttered, and provide an intuitive organisational structure that users would be familiar with using. The tree would start off showing a list of nodes on the network, and then proceed to display different 'actions' (such as inserting a file, retrieving a file), and then show the ant trails for particular fragments. There was also an additional information pane that displayed in depth detail of the item the user had selected.

The finished monitoring GUI proved extremely useful, and rapidly sped up development on the main client program.

*3) Instrumenting Monitoring in the ColonyFS Client Code:* One issue we had to consider was that the monitoring code was a potential security hole in the system. The system was

designed to hide all information about what it is doing, however someone could potentially connect a hostile aggregator tool to nodes via the interface we had provided and have access to all this data. They could then exploit the system in various ways (e.g. viewing other users' files, targeted attacks on particular files). Because of this, a design goal for the monitoring was to instrument it in such a way that all the monitoring could be easily and securely disabled when it was distributed. As the monitoring system was only intended as a development tool, this was not an issue from a user perspective. To achieve this design goal, we centralised the implementation of the instrumenting code making minimal changes to the main application code base. This allowed us to disable monitoring through a single configuration change.

## XIV. Further Development

### A. Networking Enhancements and Further Development

We have discussed several methods for further development on the system. One issue that is regularly discussed is that of security; the system is open to attack in a couple of ways easily fixed. Firstly, when a node receives a new fragment trying to replicate, it does not currently perform any checks on it to ensure it's valid. By introducing a hash check on a fragment whenever a new message is received or when a fragment is read from the fragment store, we can ensure that it has not been tampered with or become corrupted somehow. As a further security improvement we have discussed encrypting our network traffic to prevent snooping while data is in transit.

### B. GUI Enhancements and Further Development

With regard to the user experience there are several options for further development which we considered. Many of these ideas were raised during development and though desirable were not considered critical to the project. This first of these was further feature enrichment of the GUI to provide a closer integration with the native interface. This would include additional features such as automated local file deletion when a user inserts a file to the network and being able to drag items around the file tree to relocate them. We felt that while both these features would add to the user experience they were not implemented as other features were considered to be of a higher priority. Another potential development path for the project would be to provide the service as a web application. This would be achieved by running the node application on a remote server and providing a webpage through which to interact with it. We feel that this approach will give users access to their files regardless of their local computing resources including access from mobile devices. However such a development could potentially reduce the number of independent nodes present on the system resulting in one group of users maintaining a large store of fragments.

Another potential future extension involves the method of distribution. Some possibilities include embedding the system with passport file onto a small portable device such as a USB flash drive or Java Card, making usage more convenient for users. It would also provide additional security, as a physical object would be needed for access.

## XV. Conclusions

We set out to implement a distributed file system using FRS and an ant inspired optimisation algorithm, and believe we have demonstrated that these techniques can be taken out of theory and put into practice. Our work shows that it is possible to leverage an inherently unstable paradigm such as peer-to-peer to produce a robust and secure persistent file store. The success of the system relies on having enough participants willing to share their resources, however this is an assumption that underpins all peer-to-peer networks. We believe that the services provided by the system are sufficient justification for people to contribute, but accept that reaching critical mass is a potential issue.

The monitoring system we designed was extremely successful. After some initial teething problems with data processing speeds, it worked perfectly, helping to solve many problems we were having and identify new issues that we hadn't expected. We had several issues with network traffic getting lost, and the monitoring system immediately showed us where the messages were getting stuck, highlighting a problem with the network algorithm. We went through several stages of refining the networking code before we had it at a stage where all ant messages got through the system reliably, and this wouldn't have been possible without the instant feedback the monitoring provided.

We have met all of our primary objectives producing an application which enables the insertion and retrieval of data using the FRS and ant optimisation algorithms discussed in this report, while providing assurance as to integrity of the data, all of which can be interacted with via a featureful user interface. In addition to these we have also achieved several of our desirable and extended objectives. FreePastry has enabled us to provide an automated means of node discovery and allowed us to easily explore a range of configuration options to improve the system's performance. We have also addressed many of the user centric objectives including the production of a persistent structure document, the implementation of several advanced user interface features to integrate with the native operating system and the production of alternative interfaces to enable deployment to a range of platforms. Furthermore, some of the additional objectives have been partially completed or planned for, so they can be easily integrated at a later time. We are confident that this project is now at a stage where it could be realistically released for public consumption as a valid alternative for user's dependable storage needs.

## XVI. Acknowledgements

We would like the thank our project supervisors Dr. Rogerio de Lemos and Dr. Hani Ragab Hassen for all their invaluable guidance and support with the project, Dr. Fred Barnes for his assistance using the University's Pi cluster and Jeff Hoye (jeffh@mpi-sws.mpg.de, Rice University and MPI-SWS) for his help with Free Pastry. Without their contributions this project would not have been possible.

## REFERENCES


[1] W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer, "Feasibility of a serverless distributed file system deployed on an existing set of desktop pcs," pp. 34–43, 2000.

[2] S. Sen and J. Wang, "Analyzing peer-to-peer traffic across large networks," in *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*. ACM New York, NY, USA, 2002, pp. 137–150.

[3] R. Ball, J. Grant, J. So, V. Spurrett, and R. de Lemos, "Dependable and secure distributed storage system for ad hoc networks," in *Ad-Hoc, Mobile, and Wireless Networks*, ser. Lecture Notes in Computer Science, vol. 4686. Springer Berlin / Heidelberg, 2007, pp. 142–152. [Online]. Available: http://www.springerlink.com/content/r134t1t624306543/

[4] J. So, "Secure agents in a distributed storage system."

[5] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong, "Freenet: A distributed anonymous information storage and retrieval system," vol. 2009, pp. 46–66, 2001. [Online]. Available: http://www.springerlink.com/content/tmu95yypt1rd9pct/

[6] (2009, 04) Tor: anonymity online. The Tor Project. [Online]. Available: http://www.torproject.org/

[7] R. Dingledine, N. Mathewson, P. Syverson, and N. R. L. W. DC, "Tor: The second-generation onion router," 2004. [Online]. Available: http://www.freehaven.net/anonbib/cache/draft-tor-design-2004.pdf

[8] (2009, 04) Jxta(tm) community project. [Online]. Available: https://jxta.dev.java.net/

[9] A. Rowstron and P. Druschel, "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems," vol. 11, pp. 329–350, 2001.

[10] (2009, 04) Junit. [Online]. Available: http://www.junit.org/

[11] V. Roubtsov. (2005) Emma: a free java code coverage tool. [Online]. Available: http://emma.sourceforge.net/

[12] (2009, 04) Relational persistence for java and .net. JBoss Enterprise Middleware System. [Online]. Available: http://www.hibernate.org/

[13] (2009, 04) Object relational mapping, persistence and caching for java. Apache Project. [Online]. Available: http://cayenne.apache.org/