Integrating TELLTEALE as the CARROT2 IR Engine
Matthew C. Siegel
University of Maryland, Baltimore County
msiege2@umbc.edu

ABSTRACT
CARROT2 is a distributed information retrieval system. The backbone IR engine in CARROT2 is currently a piece of legacy software called Managing Gigabytes. It is the goal of this project to implement a different IR engine as CARROT2’s backbone -- a piece of legacy software called TELLTEALE. In adding legacy software to higher-level code, there are problems that can show themselves. It is crucial to make sure an integration of the legacy software is seamless, and does not end up affecting the existing high-level software. In this paper, we discuss some other legacy integrations, some of the concepts and ideas critical to legacy integration, and some tools created to help facilitate the integration of legacy systems into current day technologies.

Keywords
Information Retrieval, Legacy, Integration, Agent-based, TELLTEALE, MG, CARROT, N-gram, CORBA

1. INTRODUCTION
When working with any legacy application, it is critical to understand how it will interact with any other software it meets, including other legacy applications and newer programs based on more current technology. Bringing a legacy product into a completed system is a task that could require heavy modification of the legacy product, as many times outdated technology is not prepared or equipped to work as-is in a given job.

In this paper, we discuss other implementations of legacy systems, the problems that were encountered, and the solutions that were found to alleviate these issues. The discussion that follows details the system that we integrated, the system it was integrated into, and the method by which the integration was completed.

2. BACKGROUND/RELATED SYSTEMS
2.1 CARROT2
CARROT2 (Collaborative Agent-based Routing and Retrieval of Text), is a distributed, multi-agent information query and retrieval system. It is the goal of CARROT2 to determine if information retrieval can be accelerated by distributing the efforts across multiple nodes using java agents. Eventually, these agents may be distributed across the whole of the internet but for now the experiment resides on the ten nodes of our Beowulf system. Each agent uses a collection of metadata information in order to represent the information content of various corpora. [1]

One of the ambitions of CARROT2 is for the CARROT2 agents to be able to communicate and coordinate with other agents. They will need to interact at a high level in order to move large amounts of information from the IR engines to the other agents.

The Jackal platform is used to facilitate agent communication in CARROT2. Jackal supports conversation management abilities for the agents. By using these abilities in Jackal, we can define high level agent behaviors for management and negotiation tasks between agents. The communication language used within Jackal is KQML. Since the agents can understand KQML, CARROT2 can be accessed and configured from within another KQML based system, allowing CARROT2 to be a part of a larger system rather than just a stand-alone search tool.

When the CARROT2 system is started, there are many agents who need to register to maintain a part of the corpus. The agents talk to a collection manager, who distributes the documents within the corpus and assigns a set of documents, and metadata to each agent in the system. The collection manager agent maintains an account of which agents are responsible for which documents for search purposes.

Initially, the goal of this project is to extend existing IR engines for the agents in our system to use. In the future, it is hoped that we will be able to write wrappers to cover a wider variety of information sources such as web search engines, databases and possibly even humans.

The CARROT2 system, when originally designed, was to be used on virtually any corpus. For testing purposes, however, it was to be used with the WT10g collection of documents from the World Wide Web. This collection contains approximately 1.7 million documents from almost 12,000 servers on the WWW. The WT10g collection was loaded into CARROT2; however the MG information retrieval engine being used could not properly support the collection with an acceptable number of agents running (see section 2.3). [1]
2.2 TELLTALE

TELLTALE is an IR system that was developed to address specific concerns that were found in many other IR systems at the time of its development. It was noted by the creators of TELLTALE that most IR search tools were either traditionally based, or hypertext retrieval tools. [7] With the traditionally based tool, an entire corpus is processed, and queries are posed directly to the corpus, in either Boolean format, or natural language. In the hypertext tool situation, a user will select in a document that is linked to related information. By selecting this link, the related information is brought to the user. Each of these two types of tools has its benefits. For example, hypertext tools are less language dependant than traditional tools, and can provide a user interface making the tool easy to work with. Traditional tools allow for variability in similarly rooted search terms, by placing emphasis on the roots of words, and allowing the endings to vary. TELLTALE was designed to be a middle ground for both of these types of tools, and to incorporate the strengths of both, while attempting to eliminate their shortcomings. [7]

Developed in 1995, TELLTALE is an n-gram based IR engine. Rather than using words as the index terms, retrieval is done using sequences of \( n \) consecutive Unicode characters that are extracted from a document, or n-grams. \( N \) is usually a fixed number, depending on the corpus of data to be searched. For example, in the phrase “TELLTALE uses n-grams” the first few 4-grams found would be “TELL”, “ELLT”, “LLTA”, and so on, sliding, in this example, one character at a time through the text. TELLTALE would record each of the unique n-grams it found, and use those to match search terms, rather than unique words. If an n-gram is repeated within the text, it is given more weight within searches, rather than allowing the n-gram to be recorded repeatedly. Using n-grams, the capability of the IR engine is greatly increased. [7]

Matches may be found on only portions of a word, preventing errors in the searched documents from hindering the discovery of relevant document matches. This also is useful if a user searches on the root stem of a word, and a variation of it is found in a document. N-grams help, as well, in reducing the cost of error in optical character recognition, because incorrect characters in the middle of the text no longer cause a problem when searching. Also, users do not need to learn tedious query languages to complete searches similar to ones already completed. The n-gram approach allows for a user to search only for documents similar to a specific document. [2]

Perhaps the most convincing argument for using TELLTALE is its language independence. TELLTALE can index text in any language and has done so in English, Spanish, French, and Chinese. This use of n-grams, along with unique scoring techniques, provides the required robustness for language independence. [7] Also, because the algorithms used by TELLTALE are independent of any specific language texts to be analyzed or displayed, the corpora have no restriction on language themselves.

TELLTALE is more than just a simple text-search engine; it is also a dynamic hypertext engine. TELLTALE provides links within documents to other portions of text similar to an ‘anchor’ point. Three different types of links provide the user with the flexibility to use TELLTALE with a variety of applications. The user has the ability to use Similarity, Lookup, and Disambiguated Lookup links depending on specific needs. [8]

The goal of the project is to successfully replace MG (see section 2.3) with TELLTALE as the CARROT2 information retrieval engine. Because TELLTALE is written completely in C, and the CARROT2 agents are written in Java, TELLTALE must be wrapped in order to be used correctly. For more information about the wrapper, refer to section 3.

TELLTALE has a built in user interface. The user interface allows viewing of documents from within corpora, and shows search statistics. [2] Searches may be started from within the user interface. The user interface is completely language independent, and is capable of displaying documents in any language. While this is useful if TELLTALE was to be used on its own, the user interface is not a particularly helpful tool in our implementation, the integration with CARROT2. [8]

2.3 MG

Managing Gigabytes was written as a collection of experimental IR concepts. The power of MG is that it can create a database from a few gigabytes of text and use it to answer queries in a matter of seconds by building a complete index of every word in every document for quick reference. The database that needs to be created to accomplish this, which includes the index and the complete text, is less than half the size of the original text alone. The query actually takes less time when the database has been compressed rather than if the database is left in an uncompressed form. [14]

Originally, MG was the information retrieval engine implemented in the CARROT2 system. This legacy integration was a perfect example of some of the problems that one might encounter when integrating legacy software, namely an information retrieval engine into an otherwise functional non-legacy system.

In implementing MG as a part of CARROT2, the first major obstacle discovered was the lack of documentation that existed. While much was written regarding the concepts regarding how MG worked, the code itself was very poorly documented, and much time needed to be spent pouring over the code to properly understand not only what
it was doing, but specifically the data structures and the implementations of them that it used.

MG was designed to index static collections of data. MG will handle very well taking a corpus of data to index, large or small, indexing it, compressing it, and searching it in a reasonably swift amount of time. MG does not expect the data corpus to change. In addition, MG works best with one corpus of data, rather than many of them. CARROT2, however (see section 2.1), deals with a dynamic corpus of data.

One of the problems encountered with an unmodified version of MG was that there were many system-wide static global variables. It is these variables that caused problems with in MG. This occurred as many instances of MG were started on the CARROT2 system, each with its own corpus. The system cannot manage multiple collections of documents with one set of static variables, which is what CARROT2 needs to do. The CARROT2 team modified MG by removing these static variables, and reengineering MG to allow each instance of MG created to have its own variables kept apart from those of another instance (another agent).

Because there are many agents running on CARROT2, and each agent is assigned a portion of the total corpus, CARROT2 needs a different instance of MG for each of these agents individually. Each instance is assigned to an agent, and acts as the individual agent’s own tool to search its portion of the total CARROT2 corpus. Furthermore, as agents are added to the system, the individual documents assigned to agents change, making the specific corpora dynamic.

This behavior causes problems for MG. MG stores all of its data to disk. Once it indexes each unique word in each document, along with weights, it stores all the indices to disk. As there are more corpora, and as they change, and need to be indexed, these files must be written to disk again. Because disk writes are slow, MG causes system slowdown every time it writes to disk. With the CARROT2 system, as the number of agents increase, the frequency of updates to the MG index files also increase. As the number of agents becomes large, this causes significant slowdown across the entire CARROT2 system, hindering communication among agents. Because there is such slowdown, many of the agents timeout, and cannot complete their tasks which are vital to CARROT2’s functionality.

There were similar problems discovered integrating MG into CARROT2 regarding the metadata for each agent. In CARROT2, since metadata is distributed as a set of terms and weights for each agent’s collection of documents, it is not useful to use MG’s native method of expressing metadata. MG traditionally deals with string-based metadata. In order to resolve this, MG needed to be modified to, rather than create strings, make vectors containing the necessary metadata. In doing so, a new problem arose. Passing vectors repeatedly between MG and agents takes time, and when this is compounded by many agents doing this at once, slowdowns occurred similar to those caused by the problem with the file writes to the indices.

MG is a good example of the integration of a legacy system into a current system. The timing problems with the agents, and slow disk writes, however, were never solved. A rewrite of the file handling routines for the indices would need to have been completed and this was more complicated a task than the CARROT2 team wanted to undertake. MG did integrate well with a low number of agents running in the CARROT2 system; however, it never ran properly with the number of agents that were originally envisioned for CARROT2 to use.

3. OTHER LEGACY INTEGRATIONS

Several other integrations of legacy systems into newer software have been attempted and/or completed. Additionally, several tools exist that attempt to ease the process of integrating legacy systems.

The following sections discuss some of the systems that were integrated and the issues involved with actually implementing these systems. Also discussed are some of the tools and strategies to implement a legacy system into a newer system.

3.1 ANSAWise

ANSA, a company in the United Kingdom, has developed a set of strategies for integrating legacy applications into newer systems of today. These strategies address designing systems for flexibility while still using the legacy systems of the past. [5]

When implementing the integration of legacy applications into modern systems, according to ANSA, there are several important steps to take. Defining a role for each subsystem is a key issue. The legacy application should have set duties and it is important to understand these, and not attempt to explore outside of the scope of the application’s capability. There are well defined interfaces to each role that follow from this. It is necessary to build an object wrapper for each of the subsystems being integrated, and a common object substructure is recommended, such as CORBA in order to maintain similarity between subsystems. [5]

When re-engineering a legacy system, it is important to break the functional components of the system. Most often, the main components are not object-based, so, the biggest struggle is to build object-based replacements for each of these. Also, identification of interfaces between individual components is a key issue, as when one is
changed, it is important to note how it will affect the other interfaces in the system. In addition, rolling out each new implementation one at a time can save many headaches in diagnosing further problems in the system. [4]

ansa implements a scenario in which a fictional city, Newtown, in an attempt to encourage urban growth, wants more housing and high technology to move into the city. The Newtown authority provides access to developers to use electronic Planning and Land Registry functionality, a legacy system. The architects, however, integrate this system into a program to develop interactive design conferencing services with their clients. [5]

ansa states the most important part of the integration process is the object wrapper. Wrappers hide the legacy behind an interface. With the wrapper in place, the legacy system is only visible through the interface, and the new clients only need to be able to use that interface. CORBA, according to ANSA, is the ideal framework for legacy integration of systems.

With ANSA, we have guidelines that are good to follow. In doing what ANSA recommends, it saves the headache of having to rewrite large amounts of code, reengineer systems, and debug both the legacy and the modern systems. ANSA describes these problems which could possibly occur in a legacy integration like that of TELLTALE into CARROT2, and provides information on how to prevent them before they occur.

### 3.2 Legacy Wrapper

Legacy Wrapper, created by Tim Souder and Spiros Mancoridis of Drexel University, is a tool that packages the services of a legacy application in order to redistribute those services as a single distributed object. Legacy Wrapper allows for the use of older legacy applications in newer systems without modification to the original legacy software. Instead, Legacy Wrapper acts as a layer between the original legacy system implementation and the new program interface. In this new environment, the wrapper provides its own layer of security between the security domains of the host and the distributed object system. This security layer includes a sandbox for the application that is designed to protect the application against malicious users and the host from malicious applications. [11]

The Legacy Wrapper is designed to be a generic object wrapper. The Legacy Wrapper has 3 main components, a Legacy Client, Legacy Server, and a distributed object system linking the two, using CORBA. [11]

To test Legacy Wrapper, a dot-graph visualization tool was wrapped with it. When the graph tool is to be started, the Legacy Client starts an instance of the application; the Legacy server runs and creates a sandbox for the application to run in. In creating this example, issues with security were initially a problem, such as access rights of running the graph program; however, more important was the communication facility between the client processing the legacy data, and the server, which brought it into the context of the rest of the system. An API was later created for this. [11]

Legacy Wrapper is a tool that does, in a more generic fashion, what we are doing with TELLTALE. While it may not be as efficient, Legacy Wrapper provides methods that are important to look at, as they easily translate to the TELLTALE wrapper. The procedure by which Legacy Wrapper does all its work in a layer between the legacy application and the modern one is very similar to that used with TELLTALE, and it serves as a good base example for the TELLTALE integration.

### 3.3 Active Files

Active files is a programmable concept that ‘allows natural integration of legacy applications into distributed system infrastructures.’ The authors of active files point out that since the development of the internet, information comes in all kinds of forms, over a wide area of sources, and is ever changing. While all of these changes are occurring, most software treats files in a traditional manner. As a result, when integrating such legacy software into systems with the need to treat files and data differently, it is often necessary to either complete heavy modifications to the code of the legacy application, or write complex system-wide APIs that interface between the legacy application and the data, effectively cutting one off from the other. Both of these options perform specific high-level functions to the data before passing it to the legacy application such as filtering and format conversion. Still, such interfaces, like DCOM and CORBA, according to the developers of active files, are complex to use, and have too many restrictions, such as a completely object based legacy system. As an alternative to these options, active files is an easier to use, easier to integrate method of providing a layer between the distributed system and a legacy application.

Active files are a twist on a traditional file system. An active file is a regular file that is associated with an executable program. When a user, or another process accesses an active file, the executable linked with the active file is run as a sentinel process. The sentinel connects with the user process using pipes and has access to both remote information and the local file with which it is associated. There are both read and write pipes available for the user process to communicate with the sentinel executable. [3]

There are security holes, however, that occur when using active files. When opening an active file, the underlying file from a passive file system must technically be opened. This file is opened using the user-id of the application that opened the active file. By doing so however, there is no protection from malicious side effects that may come from...
opening the file, such as data loss, and the possibility of activating a virus. It is worth pointing out, however, that the risk of doing this is no greater than opening any other executable with a passive file system. It is possible, if security is a big concern, to implement security features for active files such as certificates and code signing techniques that would prevent some of these issues. [3]

Active files have been implemented on the Windows NT operation system. For this implementation, the active files rely on the underlying security and permissions schemes of the NTFS file system. The authors of this paper are unaware of any system that uses active files in practice today. [3]

3.4 Wells Fargo Bank

A good example of a legacy system being integrated with a newer system in a real-life situation comes directly from Wells Fargo Bank. Wells Fargo’s online banking system evolved since 1995 through the integration of older legacy components with new system functionality. In May 1995, Wells Fargo began offering real-time account balances over the World Wide Web. Since then, they have expanded their online services, now allowing funds transfers, viewing of cleared checks, examining credit card charges, paying bills, and more. In 1997, the system had over 100,000 customers, and was handling 200,000 business invocations a day. [12]

In performing the task of, over several years, adding new features, while still integrating the old system with the new, Wells Fargo modified nearly none of their legacy system. Rather, they added CORBA layerware, creating a three-tiered client server system. There is a customer object, and an account object which allows the definition of a customer relationship whereby the client can obtain all information about the customer’s relationship with the bank, and then, for each account owned by the customer, get the relevant summary information. [2]

Having, maintaining, and sharing a well-designed object model was the key for integrating the legacy system without heavy modification of old code, or the need to re-engineer a large portion of the legacy system. Rather, because the code was written in an object oriented fashion, it was easy to create a layer between the legacy system, and the modern system through the use of CORBA. In this fashion, the legacy system needs no knowledge of the higher level application for which it is being used, as the interface with the newer system was handled from the CORBA side of the system.

The Wells Fargo example is an excellent demonstration of legacy software integration. Like with the TELTALTE integration, Wells Fargo did not want to modify any of their legacy software, and they accomplished this goal. Wells Fargo shows how to go about doing so, and gives possible methods for creating the wrapper for TELTALTE.

Ideas like using CORBA and creating objects are positive starting points for TELTALTE integration. This kind of example is crucial as it can be a guideline from a project that was extremely successful.

4. INTEGRATING TELTALTE

The need for a software wrapper becomes apparent at the most abstract level when examining the IR engines we are attempting to integrate into the CARROT2 system of agents, and the system of agents themselves. The IR engines are written in C, and the agents are written in Java. Even primitive data types appear different when stored in memory from these two languages. Therefore a translator is needed in order for the agents to communicate with the IR engine.

So that agents only have to know how to use one wrapper, every IR engine we add to the system has to have a wrapper that takes the same parameters, and returns the same output. There are problems with this. Different IR engines may take completely different inputs and return different outputs so there has to be a layer of code between them and the agent to accomplish this goal. This is the second reason for writing the wrapper.

4.1 The MG Wrapper

Extended efforts have been made to implement the MG IR engine into the CARROT2 system. The current wrapper has six different functions, with at least three more pending. These are build_collection, get_meta, put_meta, get_doc, query_doc, query_meta, delete_doc, delete_meta, and put_doc.

The CARROT2 system wants to have an IR engine that not only compresses files, but completely indexes every word of text and every document within the given corpus. This makes the corpus easily searchable for meta-data. Build_collection takes as its input, a collection of documents. These are compressed, stored in the corpus, and indexed. It then outputs the index files it creates while building the corpus.

Four functions directly deal with the documents in the corpus. These are query_doc, get_doc, put_doc, and delete_doc. Get_doc takes a document id number as its input and returns the appropriate document as output. Query_doc takes a string as a query and searches the corpus for it. It returns the most relevant document along with a vector space model of other relevant documents. Put_doc simply adds a document to the corpus and delete_doc removes one.

The corpus is indexed and searchable, not only by document id numbers, but also by meta-data. Meta-data is information used by agents to represent the information content owned by various sources attached to the system, namely, the different corpora. Three functions directly deal with this meta-data. These are put_meta, query_meta, and
4.2 The TELLTALE Wrapper
Currently, efforts are underway to introduce a new IR engine, namely TELLTALE, into the CARROT2 system. In creating a wrapper for TELLTALE, we want the wrapper to be transparent to CARROT2. The CARROT2 system should work the same regardless of which IR system has been placed into it. Because we do not want to modify CARROT2, it is necessary to manipulate the data returned from TELLTALE so that it appears the same as it had been with MG. Unfortunately, the core functions of TELLTALE do not work exactly the same as those for MG. As a result, a new wrapper must be written to provide a layer between CARROT2 and the C code that TELLTALE is written in.

Since CARROT2 is a Java application, we use the Java native interface (JNI) to wrap TELLTALE. JNI is the easiest way to seamlessly call code written in other languages from within a Java program. This is a fairly common method in integration of legacy code into a Java applications, as rewriting large portions of code is much more time consuming and expensive.

To create the wrapper for TELLTALE, we first created a library of functions from the original TELLTALE system that we can call from within the wrapper. This is the legacy code that will be integrated into CARROT2. This library still contains the original code, written in C.

Of course, there is more involved with integrating the new engine than simply calling a set of old functions from within a stub framework in CARROT2. Since C and Java handle data structures differently, careful consideration must be taken in order to pass data between the C and Java layers. For example, a string in C (const char *) is allocated differently than a string in java (jstring), so within the wrapper, through the use of JNI functions, we must be careful to pass the data properly.

The functions in the TELLTALE wrapper are the same as those in the MG wrapper, however, because TELLTALE and MG operate so differently, the work done in each of these functions is not the same. In the MG wrapper, for instance, the build_collection function, discussed earlier, simply calls a function from within the MG library which sets the path to the main collection, and names it. Aside from calling a single existing function, all that needs to be done is a conversion of the two input arguments— two strings, one consisting of the path, and the other consisting of the collection name. With TELLTALE, however, three functions are required to accomplish this task. First, the path to the corpus must be set, as with MG. Following this, however, the collection must explicitly be built with a separate function. Finally, there is a TELLTALE function that must be executed to finish creation of the corpus. While this is a simple example, problems begin to arise when data structures in use become more complex, like when returning results from a query.

When a query is posed, the lookup routine in TELLTALE will return an array of integers whose length is equal to the number of documents being searched. The value of the array at any given index is equal to a weight, or document score, relative to the query posed, for the document with an id number equal to the index of the array. This information cannot be passed as-is to CARROT2. CARROT2, rather, has a Document class, which consists of a document ID, and a document weight. CARROT2 expects its IR engine to return query results to it in the form of a vector of these Documents. This is a task that needs to be performed from within the wrapper. After calling the appropriate TELLTALE functions, the wrapper must create an object array of Documents, and transfer the results from the integer array returned by TELLTALE to the object array. This format of an object array must be used in place of the vector, because JNI does not recognize vectors. When CARROT2 reads the object array, however, it will be interpreted as a vector of Documents.

Another issue that can occur with the integration of TELLTALE deals with scalability. It is not apparent how TELLTALE will perform when used with a large corpus, or when many instances (for many agents) are run concurrently. As testing of TELLTALE within CARROT2 begins, this will become more clear. Modification of the original legacy code could become necessary. Optimizations may be needed, or even possibly, reengineering of certain portions of the could might be required. That is, data structures used in TELLTALE may not be appropriate for use in CARROT2, and new methods may need to be created to keep the same functionality during integration.

5. FUTURE WORK
Once the TELLTALE wrapper has been completed, there will be extensive testing done on the CARROT2 system, as this will be the first real chance to operate the system with a substantial number of agents running.

Other plans include using CARROT2 to search the World Wide Web. A wrapper will also need to be written for this purpose, to serve as a layer between the CARROT2 agents, and the documents in existence online. This is a step in the direction of an agent-based information retrieval system on an extremely large-scale, extremely dynamic searching environment. While it is unclear when or if these goals will be achieved, they are in the future plans for the CARROT2 system.
6. ACKNOWLEDGMENTS
The work on CARROT2 and TELLTALE is sponsored by the Department of Defense.

Our thanks to ACM SIGCHI for allowing us to modify templates they had developed.

Jonathan Kessler contributed to portions of this paper.

7. REFERENCES