IST-2001-32133
GridLab - A Grid Application Toolkit and Testbed

Grid Application Toolkit Canonical Implementation and User's Guide

Author(s):	Kelly Davis
Document Filename:	Gridlab-1-CGUG-0013.CanonicalGATImplementationUsersGuide
Work package:	Grid Application Toolkit
Partner(s):	PSNC,ZIB,MU,SZTAKI,VU,ISUFI,CARDIFF,GRIDWARE, COMPAQ,NTUA
Lead Partner:	MPG
Config ID:	Gridlab-1-CGUG-0013-1.0
Document classification:	Internal

Abstract: This document describes the Grid Application Toolkit (GAT) implementation and guides one through the use of the GAT, covering everything from high-level architecture to the details of installation and use of the toolkit.

Information Society Technologies

Contents

1 Preface

1.1 Why GAT?

Why is there a need for a GAT, the **G**rid **A**pplication **T**oolkit? The simple answer, change.

The "Grid" currently is part-research, part-reality, and anything but settled. The "Grid" currently consists of a set of semi-related technologies, each of which is also in a state of flux, that allow users to utilize resources in a decentralized manner. As these various technologies are currently in a state of flux, the hapless application programmer, who's task it is to build an application for these users upon these shifting sands, is yolked with a Sisyphean task. GAT removes the application programmer from this quagmire. GAT isolates the application programmer, who only wishes to make use of the "Grid," from the shifting sands of the "Grid's" ever changing shore.

1.2 Organization of the User's Guide

This user's guide can be divided into two sections.

1. Chapter 1 indicates the scope of this user's guide.
2. Chapters 2 through 12 make up the core of this user's guide. These chapters presents the core functionality of GAT as seen from an application programmer's point-of-view.

The chapters within the second section need not be read in order. However, usually chapters within the second section build upon one another. Thus, Chapter 5 may use something from chapter 3.

1.3 Who You Are

This user's guide assumes that you have a basic familiarity with the C89/C99 programming language, have a programming environment for the C89/C99 programming language, and a basic familiarity with the concepts of object oriented programming. This user's guide does not attempt to be a basic C89/C99 language tutorial; you *should* be familiar with the C89/C99 programming language.

You should be comfortable using the Internet. This user's manual assumes that you know how to visit web sites, download files, and uncompress downloaded files. In addition, you are assumed to have a basic knowledge of what a URL is and how to go about examining the resource to which a http, say, URL points.

1.4 GAT Versions

This user's guide is developed using the 1.0 version of GAT.

1.5 About the Examples

Almost every function in this user's guide is illustrated with at least one complete working program. This allows you, my humble reader, to experience first hand what the function's documentation actually means in practice. The language of code is much less forgiving then the natural language of English. Hence, any topic which is badly explained by the GAT documentation, and yes there are some (gasp!), or badly explained by this user's guide, yes there are some of these also (double gasp!), will hopefully be rendered crystal clear through the use of code samples.

The end of each chapter in the third section contains at least one, and more commonly several, programs which demonstrate the use of the various functions covered in the chapter in more detail. These present the various functions covered in that chapter in a more "realistic" setting, though even then we often skimp on the error handling as in C89/C99 error handling tends to clutter the code with endless `if` statements; we will hopefully target Java and C++ in future releases. All the example programs are available online, often with corrections. To save you from typing, they are available from the GridLab web site, *http://www.gridlab.org*.

All examples assume you are using a C89/C99 compatible programming environment. All examples here should work in such an environment. However, some vendors may have cut-corners on their C89/C99 implementations. If you find yourself in under the yoke of such an implementation, well, I wish you luck.

1.6 Conventions Used in the User's Guide

This user's guide uses the following conventions:

A `constant width font` is used for:

- Anything that might appear in a C89/C99 program.
- Path names, file names, and program names.
- Command lines and options that should be typed verbatim on the screen. A **`bold constant width font`** is used to indicate the output of a command.

An *italic font* is used for:

- New terms where they are defined.
- Internet addresses, such as domain names and URL's.

When code is presented as fragments, rather then complete programs, the existence of the appropriate `include` statements should be inferred. For example, in the following code fragment you should assume that the header *stdio.h* was included.

```
printf("Hello world!");
```

1.7 Request for Comments

I would like to hear from readers about how this user's manual can made better. With the limited time I had in which to write this user's manual, I am sure there are many errors which have crept into its pages. So, please let me know where they are. My email address is `kdavis@aei.mpg.de`. Please realize, as I am likely on some other barn-burner, that I may not have time to answer each and every email.

1.8 GAT Installation

Refer to the GAT Installation Guide for installation of GAT.

1.9 GAT Requirements

To install and use GAT you should have the following:

- A C89/C99 programming environment.
- A GNU compatible make system.
- A Linux-like command line environment.

2 Why GAT?

2.1 Why GAT?

Beginnings are sacred things. So, with this in mind, we begin this user's guide with a "why" instead of a "how." "Why's" are always more illuminating in my opinion, and "how's" are a dime a dozen.

So, we begin. Why GAT? The answer, in a word, is simplicity. The "Grid" is currently part research project, part reality, part hype, and everything but settled. The "Grid" is a word used for a collection of various semi-related technologies which allow a user to utilize decenterlized resources. The problem with the current state of the "Grid" is that this loose collection of semi-related technologies does not at present present a uniform interface to programmers. *The* "Grid" technology of today is all but forgotten tomorrow. For example, one of these technologies may be written in C and expect to be called using XDR protocols; a second technology may be written in Java and expect to be called using RMI; while a third technology may be written in C# and expect to be called locally. For the application programmer, the poor soul who is stuck trying to integrate all of these desperate threads, life in such a world is, let us say, less than amusing.

Application programmers are stuck between the devil and the deep blue sea. The users foist upon the application programmer their expectation of being able to use an endless ocean of resources. If the users are presented with anything less then the "Grid" hype about which they have read so much, they berate the application programmer pointing out the chasm between expectation and reality. However, if the application programmer tries to actually breath life into the "Grid" hype, they are tied to the rack of learning a myriad of differing technologies. GAT hoists the application programmer out of this Catch-22.

GAT presents to the application programmer a uniform interface to "Grid" technologies. The actual "Grid" technologies which implement the GAT API functionalities are plugged into GAT by means of a plugin architecture. This allows the application programmer to worry about the application and the "Grid" technology experts to worry about the plugins which talk to their services over whatever protocols they have dreamed up. The idea of GAT is simple, yet powerful.

2.2 What Can GAT Do?

As the "Grid" consists of a pied collection of semi-related technologies, each of which allows the user to utilize decentralized resources, one might gather that any uniform interface to such "Grid" technologies might also present a piebald set of functionality. This is indeed the case.

The uniform interface which GAT presents to "Grid" technologies allows the user to carry out any number of various tasks. . .

- The user can engage in file management – she can move files, copy files, delete files, and examine various file properties – all in a manner independent of the file's location or method of access.
- GAT presents to the user a means of streaming to and from files – she can read, write, and seek on file streams – in such a ways as to lift the burden of protocol management, security, and various other details from the user's shoulders.
- The user can efficiently replicate files – she can use the logical file management utilities to replicate files so as to maximize use of available network bandwidth – and never have to worry about the details of security, protocols, Globus. . .
- She can utilize a classified advertisement like system which allows application to share instances and associated data across process, machine, and organizational boundaries. In addition, she is never yoked with the details of such process. It just works.
- The user can pursue the task of resource management – she can find and reserve hardware and or software resources – all the while being blissfully ignorant of the machinations involved in such priestcraft.
- GAT allows for the user to communicate across process boundaries – she can write and read bytes to remote or local processes – while remaining ignorant of the underlying protocols used to achieve this goal.
- Job management is facilitated by GAT – through GAT, users can schedule jobs, unschedule jobs, stop jobs, checkpoint jobs, migrate jobs, and any number of other useful things – while never bothering the user with the details of traditional job management systems such as Condor, Globus, Unicore, . . .
- In addition, GAT allows for the user to monitor the vast majority of the various machinations being carried out in the user's name.

However, as "Grid" contains such a miscellany of technologies within its expansive boundaries, we sadly couldn't pack it all in GAT. In creating GAT the decision was made to put some things in and leave other things out. That which is in GAT represents the functionality which we saw as most useful to the "average" application programmer. That which is not in GAT represents what we saw as the functionality which is only marginally useful to the "average" application programmer. So, as in all things, everyone will have their belle de jour which, somehow, was left out of GAT, but the majority of application programmers will have the majority of their needs met the majority of the time. So, what can this GAT thing do really, the details this time. . .

2.2.1 File Management

GAT allows one to manage remote and local physical files in a manner which is independent of where the physical files are and the protocols used to access such physical files. This is a particularly powerful, yet simple concept. An application programmer can write an application which manages physical files (moving, copying, deleting, . . .) and the application programmer never has to worry about if the physical files are accessed via *http*, *https*, *ftp*, or any other obscure protocol. The application programmer never needs to worry about if the remote FTP server is up, if the local version of Globus is compatible with the remote version of Globus, or if the stars are in the right house of the Zodiac for the physical file transfer to take place. The application programmer is blissfully ignorant of such details. File management just works, as it should be.

Creating and Deleting Files The bread and butter of physical file management consists of creating and deleting physical files. GAT, of course, allows one to create and delete physical files. GAT does so in a manner which allows users and application programmers to live in blissful ignorance of the machinations going on under the covers. To the application programmer and user alike, all the resources to which they have access look like a giant hard drive on which they can create and delete physical files to their heart's content. It just works, as it should be.

Coping and Moving Files After creating and deleting physical files, the two most important physical file operations are copying, and moving physical files. GAT, naturally, allows also for both of these operations. GAT also, naturally, allows for both of these operations to be completed in a manner independent of the operating systems and protocols used to access the various physical files. The same code can be used to copy a physical file from a Linux machine to a Solaris machine using grid ftp or to copy a physical file from a Windows machine to a OS X machine using XML-RPC, or the same code can be used to move a physical file from a AIX machine to a BeOS machine using http POST or to move a physical file from a PalmOS machine to a PlayStation machine using a WSDL based service. The user just specifies the "from" and the "to" and GAT takes care of the rest, security, permissions, and protocols, a package deal.

Examining Files In addition to creating, deleting, copying, and moving physical files GAT allows for the application programmer to examine various physical file properties. The application programmer can examine if a physical file is readable or writable. She can get the length of a physical file in bytes, get the time at which the physical file was last modified, and determine if two physical files are "semantically equivalent." Again, GAT hides the application programmer from the (edward) gory details of all this. The programmer simply queries GAT for the apropos information and GAT obeidently replies.

2.2.2 FileStream Management

GAT allows one to write and read from a remote or local physical file independent of where the physical file is, independent of the protocols used to access the physical file, and independent of the OS which is managing the physical file. This again seems particularly simple as an idea, but the result of embracing this simple idea is rather powerful. One can read bytes from a physical file on an XBox with the same code used to read bytes from a physical file on a PlayStation, or a AIX box for that matter. GAT isolates the application programmer from the details involved in talking to an XBox, or a PlayStation, or an AIX box. All the application programmer sees is the functionality they need exposed in a crystal clear, transparent manner.

Creating and Destroying FileStreams The core of physical file stream management consists in creating and destroying "FileStreams." *FileStreams* are the abstractions in GAT introduced to allow the application programmer to read and write bytes to an open physical file. GAT, not surprisingly, allows for the creation of such abstractions and their subsequent destruction.

Reading and Writing on FileStreams Upon creating a FileStream the question next arises: "What can one do with a FileStream?" ("I got three nickels and a quater, can I get to El Sugundo?") Among other things one can read and write bytes to the FileStream. This will in turn read and write bytes to the physical file represented by this FileStream. The magic of GAT occurs in that the application programmer need not be concerned if the FileStream corresponds to a local physical file or a remote physical file or if the protocol used to access the physical file is "the" newest kid of the block. If the physical file is local, the application programmer writes the same code as if the physical file were remote. If the file is accessed using ftp, the application programmer writes the same code as if the physical file were being accessed using https. GAT, the magic happens here.

Seeking on FileStreams In addition to the ability to read and write on FileStream's GAT allows for the ability the seek on FileStream's. In other words, whenever one is reading from or writing to a FileStream one is reading from or writing to a particular place in the physical file corresponding to the FileStream. For example, one may be reading such that the next byte is the 367th byte in the physical file. Or, for example, one may be writing such that the next byte written will by the 763rd byte in the physical file. In both cases there is a "pointer" which indicates where the next read or write in the physical file will occur. Seeking in the act of moving this little internal pointer. ("Ok, jump ahead 23bytes, now back 267, finally go to byte 1741.") So, for example, I can be reading byte 367 of a physical file using a FileStream, then decide I want to read byte 123 as the next byte. The GAT FileStream allows one to do so by *seeking* on the FileStream. As you might have guessed at this stage, GAT frees the application programmer from the need to worry about if the physical file corresponding to the FileStream is being accessed through TCP, UDP, http. or any other four letter acronym. One asks of GAT and low yea' shall receive.

2.2.3 LogicalFile Management

It is often the case that when working with large physical files in a geographically distributed computing environment that one has various physical files which are byte-for-byte identical, but yet are distributed geographically. When this relatively common situation occurs, deciding which of these geographically dispersed large physical files to use for a particular operation becomes a problem. For example, if one decides to copy one of these physical files to a computing resource on which it does not currently exist, then choosing the "wrong" copy can cause a wait of hours as this large physical file is copied through a connection which is already at its saturation point. GAT deals with this very situation by introducing the LogicalFile construct.

A LogicalFile, see figure 1, represents a set of physical files which are byte-for-byte identical but geographically dispersed.This construct is useful for the very case described above, as well as other related cases. A LogicalFile representing the various physical files in the previous example can be used to facilitate the above copying. However, the actually decision as to which of the various physical files to use to make the copy is left up to GAT and its minions; they decide

Figure 1: A LogicalFile can be thought of as a folder containing many identical files.

which of the physical files is closest in "network space" to the target location, then use that physical file to make the new copy so that the process is done as efficiently as is possible.

Creating and Destroying LogicalFiles GAT, of course, allows for the creation and destruction of LogicalFiles, as this is the most fundamental operation involved with LogicalFiles. This is done using standard language constructs. If you couldn't create one of these, it wouldn't be very useful now would it?

Adding and Removing Files Upon creating a LogicalFile one needs to indicate which physical files are represented by this LogicalFile. This is done through the process of adding and removing File instances to a LogicalFile instance. The process is just like adding more files to your office folders, see figure 2. But the difference is that in adding File instances to a LogicalFile the user and application programmer should execute caution and only add files that are byte-for-byte equivalent to any other files which are already present in the LogicalFile. For you office folders, though, you can add whatever you want. If one does not heed this caution, then the LogicalFile instance will end up being a headache instead of a help, as you'll have all your apples mixed in with your oranges.

Replicating LogicalFiles The primary use of the LogicalFile construct is for replication. If one wishes to replicate a physical file represented by a LogicalFile instance to a new location, then one simply uses the LogicalFile to do so. GAT through much practice at prestidigitation determines which of the physical files represented by this LogicalFile is closest in "network space" to the target location and uses that physical file to make the replica. The legerdemain of the entire process is that the application programmer and user are completely unaware of the detailed process involved in choosing which of the various physical files to use for the replication. Network bandwidth, number of network hops, permissions, and a myriad of other esoteric network parameters must be divined in order to read the tea-leaves of the network's future and determine the "best" physical file to employ. The application programmer simply chants a single incantation "replicate" and the theurgy proceeds.

Figure 2: Adding and removing Files for a LogicalFile is analogous to adding and removing files from folders. However, when adding and removing files in folders, one need not make sure all the files in a given folder are duplicates, but this is not the case for a LogicalFile.

Examining LogicalFiles On a more earthly note, GAT allows one to examine a LogicalFile instance. In particular, one can determine which physical files, in other words File instances, are represented by this LogicalFile instance. This is extremely useful for the case in which one obtains a LogicalFile instance from a third party, especially if they are "shady," as one can never really know what physical files are represented by a stranger's LogicalFile instance unless one looks.

2.2.4 Advertisable Management

Within real "Grid" computing it is often the case that one needs to make a particular object persistent or one needs to allow a particular object to be accessed from another computer. The "advertisable" management system makes this possible. More than that, it make it easy!

Assume for the moment you have a run-down car to give away, aone red 1973 VW Beetle, and you wish to place an advert in the paper for this advertisable car. The first thing you would do is write a few sentences describing the car. Basically, these sentences could be summarized by a set of name value pairs "color=red", "year=1973", "type=VW Beetle", "price=free"! One would then contact the newspaper that one wishes to place an advert in. This newspaper provides the service of hosting adverts; so, it can be thought of as an advert service. Upon contacting the newspaper one places the advert by associating the name value pairs "color=red", "year=1973", ... with the physical car in a forum that the public can access, see figure 3. Using the advertisable management system in GAT is no different from placing this add in the newspaper. It's so simple you know how to do it already.

Assume now that you have a File you wish to advertise. Assume it's a mov file which is 3.9 MB. Using GAT you first create some "meta-data," a fancy word for data about data, describ-

Figure 3: Typical newspaper advert service.

ing the file. This meta-data is captured by a set of name value pairs, "type=mov", "size=3.9", "size.units=MB", which are placed in a Table. One then contacts the AdvertService and places the advert buy using the AdvertService to associate the Table with the File in a forum that the public can access. Any other user can then contact the AdvertService and query it, again using some meta-data, to obtain your File. Just like giving away the old VW for parts.

Creating and Destroying an AdvertService GAT, needless to say, allows for the creation and the destruction of and AdvertService. This process of construction and destruction is accomplished using standard language constructs; so, we won't trouble you here with this piddling details. Later though, we'll drub you with the details.

Adding and Deleting Advertiseables Upon creating an AdvertService the first thing usually wants to do is to add an Advertiseable to, or delete an Advertiseable from this AdvertService. To add an Advertiseable to a AdvertService the first thing one does is to create some meta-data describing the Advertiseable. This meta-data is captured by a set of name value pairs placed in a Table. After populating a Table with this meta-data, one then simply associates this meta-data with the Advertiseable through a single call to the AdvertService. Upon completion of this call, the Advertiseable is placed in the AdvertService and other users of GAT can access this Advertiseable by simply querying the AdvertService. In addition, if for some reason one realizes that placing this advert was a huge mistake, one can just as easily remove the Advertiseable from the AdvertService.

Finding and Obtaining Advertisables Beyond just adding and removing Advertiseables from the AdvertService one can also query the AdvertService to determine what Advertiseables it contains and then obtain the desired Advertiseables from the AdvertService. This is done by creating a Table containing meta-data describing the Advertiseables you wish to find, then querying the AdvertService with this meta-data. With this meta-data, one can perform a search to obtain these Advertiseables in a simple, wax-on, wax-off procedure.

2.2.5 Resource Management

The "Grid," at least in theory, consists of a huge set of resources: OS's (From Plan 9 to OS X), hardware (From PalmPilots to the Earth Simulator), software (From "Hello World!" to climate models). This huge multifariousness leads to the obvious quandary. How does one find what they want? Given a very limited set of resources to which we have access, the problem does not exist. (Henry Ford was the obvious champion of this mode of operation. "You can have any color car, as long as it's black.") However, the "Grid" allows users to access and almost endless ocean of resources, see figure 4. As a "Grid" user, we may have access to resources we didn't even know existed. Thus, we have overcome the blinders of Mr. Henry Ford only to be blinded by the vastness of our current choices. We must now address the question of, "How to choose?"

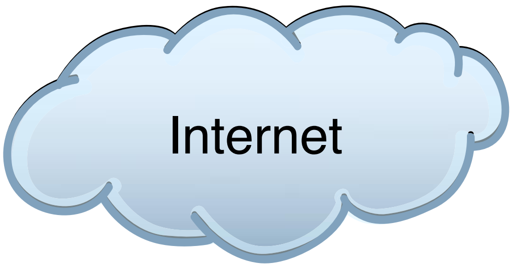

Figure 4: The Grid, the resource cup floweth over.

From the depths of its ocean trench, GAT swims to the rescue with a common theme in this guide. GAT has several abstractions which enable users and application programmers to leverage a vast sea of resources. The "Big Fish" in the school of GAT tools is the ResourceBroker. A *resource* is any element providing some capability and a *broker* is an agent who distributes or aggregates resources on the behalf of another. So, as one might guess, a ResourceBroker is a software abstraction that distributes or aggregates resources. If one wishes to find a resource, be it software or hardware, one simply queries a ResourceBroker to find the apropos resource. Similarly, to reserve a resource one does so using a ResourceBroker as an intermediary. Lets look at some of the details.

Creating and Destroying a ResourceBroker Of course GAT allows for the creation and destruction of a ResourceBroker. There are, however, many other functionalities which one can access through a ResourceBroker.

Creating and Destroying a ResourceDescription The first step in querying ResourceBroker for a resource is describing the resource you want. For example, you might decide that you want a hardware resource running the operating system Darwin. This can be expressed by specifying a name and value pair, "os.name=Darwin" say. In point of fact, this is exactly how GAT functions. One creates a ResourceDescription, using standard constructs, that contains a well-defined set of name value pairs that describes the hardware or software resource that one wishes to depict. This ResourceDescription can then be used to query the ResourceBroker or it can simply be destroyed.

Find Resources Finding a resource is a fours step process, see figure 5. First one creates a ResourceBroker; second one creates a ResourceDescription describing the resource one wishes to find. Next, one calls a single function on the ResourceBroker passing it the ResourceDescription describing the resource one wishes to find. Fourth the ResourceBroker then does all the "heavy lifting," querying the "Grid" for all the various resources which fit the bill and returning a list of such resources to the caller. Simple!

Reserve a Resource Using a ResourceDescription In addition to simply finding resources one can also reserve resources. This process is similar to the process of finding a resource. One first creates a ResourceBroker then one creates a ResourceDescription describing the resources one wishes to reserve. Next one must specify the time period one wishes to reserve the resource. This is accomplished by creating a Time instance, which indicates when the reservation is to start, and a TimePeriod instance, which indicates the duration of time the resource is to be in use once the reservation is placed. Finally, one passes the ResourceDescription, Time, and TimePeriod to the ResourceBroker. Upon obtaining all of this information the ResourceBroker marshals all of its minions to scour the "Grid" for a resource which fits the bill. The user is completely unaware of the machinations which are going on behind the curtain, as it should be. They just get the results, a reservation, and go on their happy way.

Reserve a Resource Using a Resource Another method of reserving resource also exists. Upon finding a resource, as described above, one obtains a Resource instance. This Resource can be used in place of the ResourceDescription in the reservation process described above. So, one would pass a ResourceBroker a Resource instance, describing the resource to reserve, a Time instance, which indicates when the reservation is to start, and a TimePeriod instance, which indicates the duration of time the resource is to be in use once the reservation is placed. The result is the minting of a brand new Reservation.

2.2.6 Interprocess Communication

Up to this point the only manner in which GAT facilitates interprocess communication is through the AdvertService. If this were the only means of interprocess communication GAT offered, GAT would be hobbled at the knees and deserve to be put down. GAT, however, offers more than this.

Talking on the phone with someone is something everyone is familiar with. ("Hey, Joe. What's up?" "Nothin' Ed." "Joe, you got any of them *real* clean white shirts?") You obtain a phone connection from the phone company, and, if you want them to, they list your phone number in the phone book. Other people interested in calling you can look up you phone number in the phone book and call you. If you are home when they call, the two of you can talk.

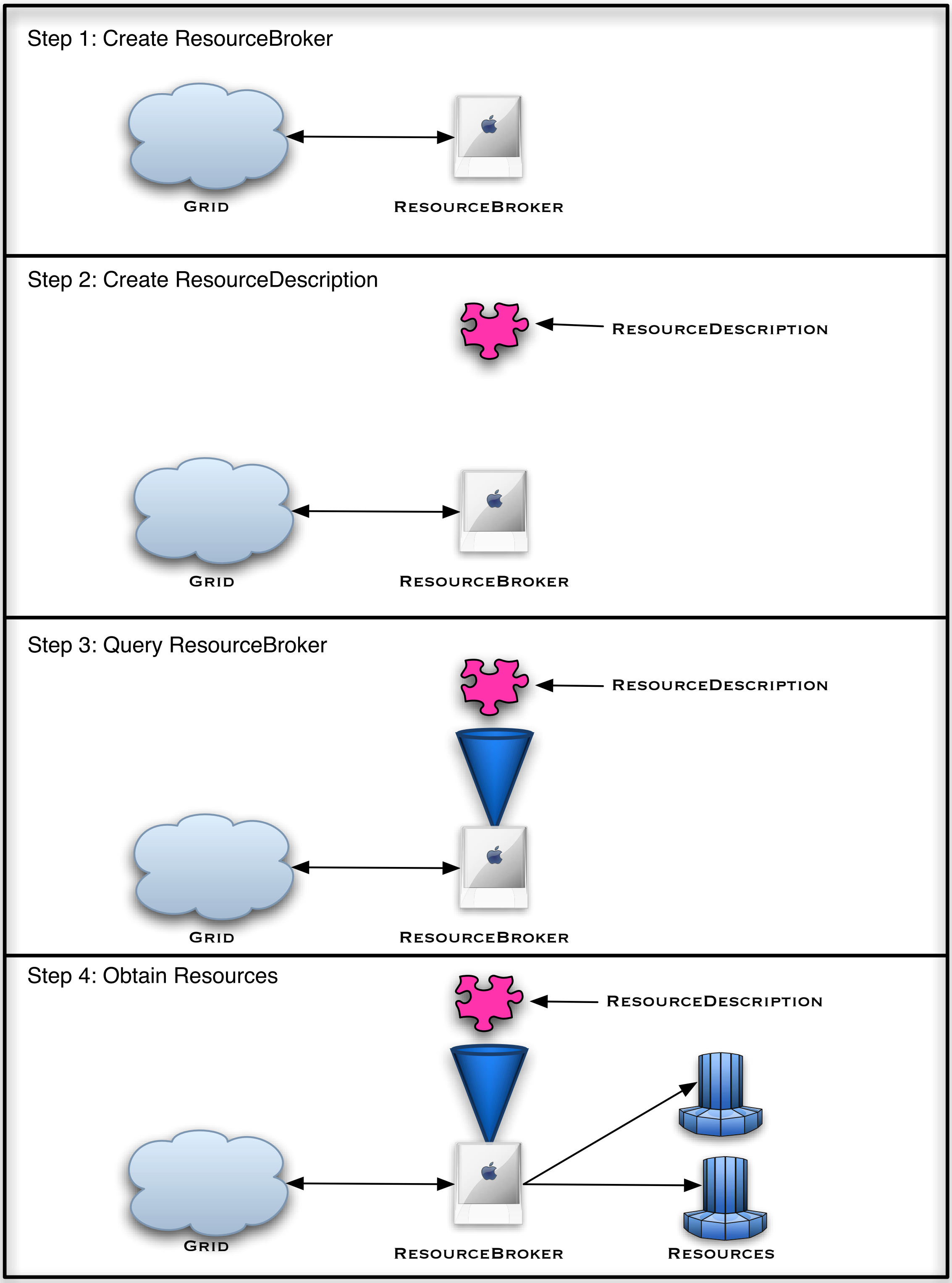

Figure 5: The steps in finding resources.

This simple model of a phone call also extends to the model of interprocess communication used by GAT, see figure 6. A process, if it wishes to be contacted by other processes, creates an Endpoint and places this Endpoint in an AdvertService. (This Endpoint is the analog of a phone number and the AdvertService is the analog of the phone book.) This process then "waits by the phone" "listening" for any "incoming calls." If a second process wishes to contact this first process, it looks up the first process' Endpoint in the AdvertService, removes it, and uses it to try and open a channel of communication between the two processes. If the first process is "waiting by the phone" when the second process "calls," it receives the "call" and a communication channel is established through which the two processes can exchange data. The model is relatively straight forward. Lets look in a bit more detail at the various abstractions involved.

Creating and Destroying Endpoints An Endpoint is the primary abstraction used for interprocess communication. . In a general sense, an Endpoint represents one end of a "phone line." Specifically, an Endpoint represents one end of a byte stream over which data may flow. Creation of an Endpoint proceeds through standard mechanisms. The user does not have to specify their IP, their operating system, or any other such niggling particulars. GAT keeps track of such silliness internally. Destruction of an Endpoint is equally painless.

Advertising Endpoints Upon constructing an Endpoint we need to place it "in the phone book" so people can actually "call" us. What's the point in having a phone if no one call talk to us? To make our Endpoint known to other processes, we place it in the AdvertService. As an Endpoint is Advertisable, the steps involved in placing it in the AdvertService are the same as for any other Advertisable: Create meta-data associated with this Advertisable, a Table instance, and associate this Advertisable with the meta-data through a single call to the AdvertService. By placing this Endpoint in the AdvertService we've primed the pump for other processes to "call us up."

Listening on Endpoints Before you can get a call, you have to listen for the phone ringing. Likewise, before you can establish a communication channel with another process, you have to listen on an endpoint. In the case of a real phone you listen. In the case of and endpoint, you call the `Listen` function.

Connecting on Endpoints Now to place a call, you need a phone number. So, if, for example, you were looking for your nearest antiques dealer so you could get your "Land of the Lost" fix by purchasing a super groovy 1975 sleestack costume, made with loving care by Ben Cooper, then you would look in a phone book to find the nearest antiques dealer which dealt in "Land of the Lost" propaganda. Similarly, if you were looking to contact a particular process using GAT's interprocess communication mechanisms, then you would find and remove the corresponding Endpoint from an AdvertService.

Once you have a phone number to place a call you have to push those little numbered buttons to finally make a connection. Likewise, after obtaining an Endpoint from an AdvertService, one must take one final step before one can contact a second process. One must call the function, cunningly called, `Connect`.

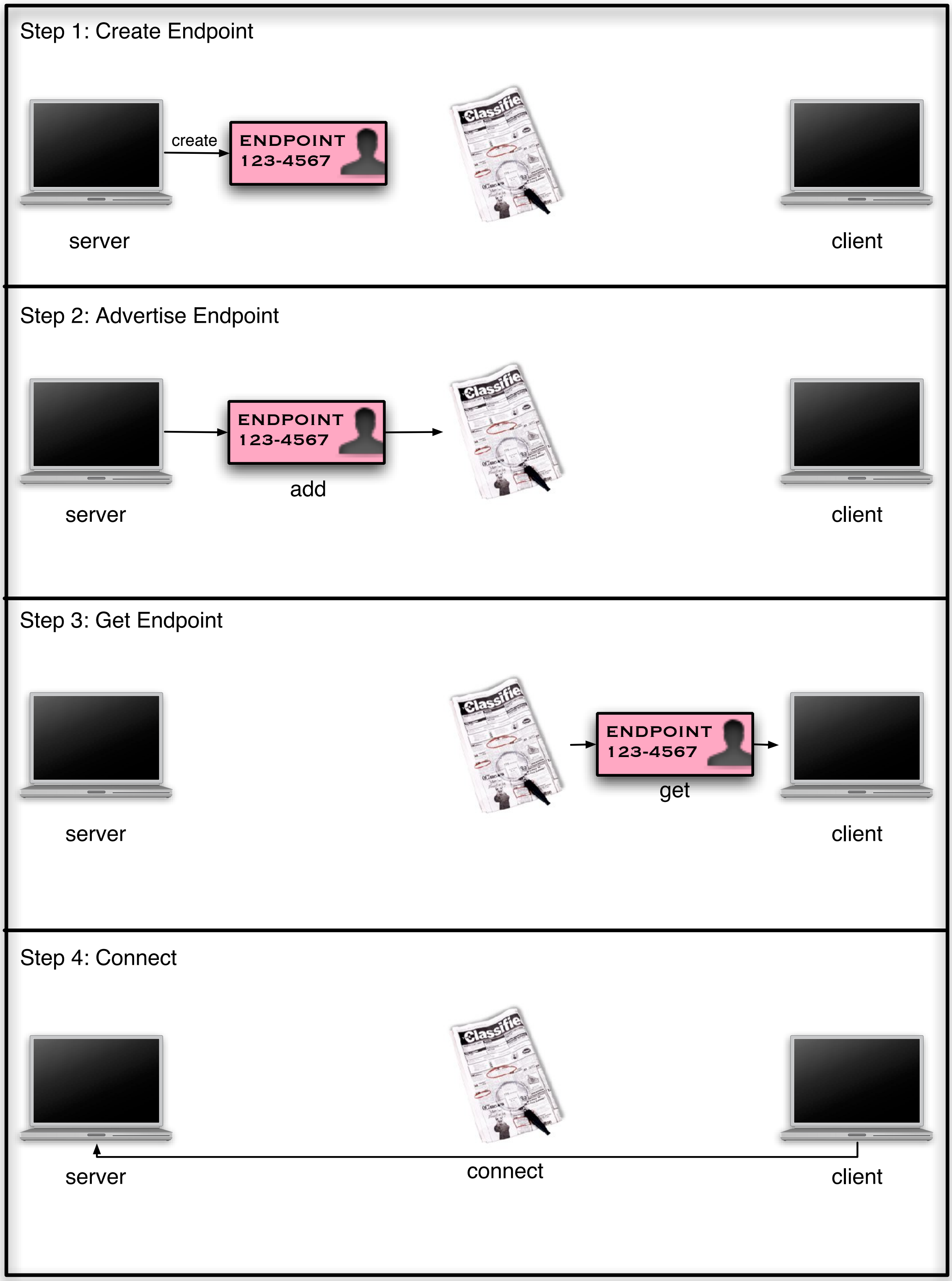

Figure 6: The steps in interprocess communication.

Write and Read on Pipes Now that you've got the phone number, dialed the phone, and waited for the other party to pick up, you still have to say something. For example, your "jonesin"' for the ever popular "Land of the Lost" 1975 sleestack costume can't be satiated by simply calling up the nearest antiques dealer. After they answer the phone you have to actually talk to them! Likewise, in the land of the GAT, after a process has called the `Connect` or `Listen` function, they must actually send or receive data. In a call to the function `Connect` or `Listen` one is returned a Pipe instance. Using the `Read` and `Write` functions on this Pipe instance one can read from and write data to the remote process, and get your "Land of the Lost" fix.

2.2.7 Job Management

One of the key uses – if not "the" key use – for this nascent "Grid" is "job management." By job management we simply mean the stopping, starting, checkpointing, migrating, . . . of computer processes on apropos "Grid" resources. Imagine writing an accurate simulation of the human brain which models all structure from neurotransmitter to encephalon from rhombencephalon to prosencephalon in BASIC on your Commodore VIC-20 then trying to run this code art only to realize that your VIC-20 is a hobbled gimp with only 3.5K of RAM available to BASIC programs. Naturally, you'd want to execute your beatific code somewhere else, somewhere more appropriate, somewhere where the "little me" you've cultured can be set free from the confines of 3.5K. The question is: "How to release your Galatea from the base stone of 3.5K of RAM?" The answer is: "GAT."

GAT's job management system allows users and application programmers alike to effortlessly launch processes on apropos "Grid" resources and subsequently manage such processes. All the application programmer or end user need do is describe in a very general way the software they wish to run and the hardware they wish to run it on, and GAT does the rest. So, for example, to run this encephalon on an apropos resource all you need do is describe the resource you want to run on, "A Linux box with 32000000 processors," and describe your Galatea, then tell GAT about both. GAT will find such a resource for you, if it exists and you have permissions to use it, then start your artful code on this beast.

Creating and Destroying Jobs To create a Job is a bit more involved then creating other GAT instances, see figure 7; in creating a Job you must take at least two steps back before you can take one forward. To create a Job you must first create a JobDescription. To create a JobDescription you have to create a SoftwareDescription, describing the software the job will run, and a ResourceDescription, describing the hardware the job will run on.

Creating a SoftwareDescription is rather straight forward. One simply has to describe a bit of software, then package up this description in a manner GAT can grok. In the case of a SoftwareDescription GAT grocks name value pairs. One creates a Table containing a set of name/value pairs, the universe of names and values are specified in the GAT API specification, describing the software one wishes to run, then one uses this Table to construct a SoftwareDescription. A similar story is also true for creating a ResourceDescription. A ResourceDescription is also constructed with a set of name/value pairs, the universe of which are specified in the GAT API specification. With these two descriptions described we can move on to the final description, JobDescription.

As everybody knows, except for Fred who's always sleeping in the back of the classroom, to run a job you need to specify the software to run and what hardware to run it on. So, to create a JobDescription you need to specify a description of the software you need to run and a description of the resource you need to run it on. Cunningly, you describe the software with a SoftwareDescription and the resource with a ResourceDescription. With an instance of a SoftwareDescription in your left hand and an instance of a ResourceDescription in your right you can create a JobDescription lickety-split by just passing both of these to the JobDescription constructor.

Finally, the destination is in sight we can create a Job by passing a JobDescription to the function `SubmitJob` on a ResourceBroker instance. Destroying is much simpler, simply call Destroy on the Job.

Examining Jobs Now we have a job, what do we do? First lets take a poke around and find out a bit about our Job. The obvious place to start would be to get a description of our job. This is easy, a call to the Job function `GetJobDescription` returns just that. Also, one can examine the state, initial, scheduled, running, . . . , of our job with a simple call to the deceivingly named Job function `GetState`. Also, if you still are itching for more info, then you can call on the Job function `GetInfo` which returns a Table containing a set of name/value pairs describing your job.

Scheduling and Un-Scheduling Jobs Before talking about how to "schedule" or "un-schedule" a job, lets take a step back and talk about what "scheduling" and "un-scheduling" are. Scheduling a job is the process of adding a job to a queue so that it may at some future time start running. Think of it like this, you're living in the early 30's, depression area, have no work, and have thread-bare clothes. So, you, along with almost everyone else, is waiting in a queue for work. You wait and wait and wait until finally you get to the front of the queue and then can begin working. It's just like that for the job, but of course it'd have far better looking clothes on. Un-scheduling is the process of removing this job from the queue. Now that we have that out of the way, we can talk about how GAT facilitates these processes.

First scheduling. During the process of creating a Job it is automatically scheduled. That was easy. So, in particular, one shouldn't play about creating jobs, as when one creates a job it's "live" and actually using resources on some real computer. To un-schedule the job one simply calls the function `UnSchedule` on the Job instance. Nice how the responsibility for scheduling and un-scheduling is spread across two classes. This is just a little quirk of `GAT` that I'm sure will be ironed out in time.

Stopping Jobs Stopping a Job is much easier then stopping say Dr. Strangemind from *Lancelot Link Secret Chimp*[1] To stop a job all you need do is call the `Stop` function. To stop Dr. Strangemind. . . well I don't think even Lance knows the answer to that one.

[1] *Lancelot Link Secret Chimp* was a TV show in the 70's which followed the exploits of a secret agent as he battled the likes of Dr. Strangemind and other arch villans. Oh yeah, I almost forgot, Lance was a chimp.

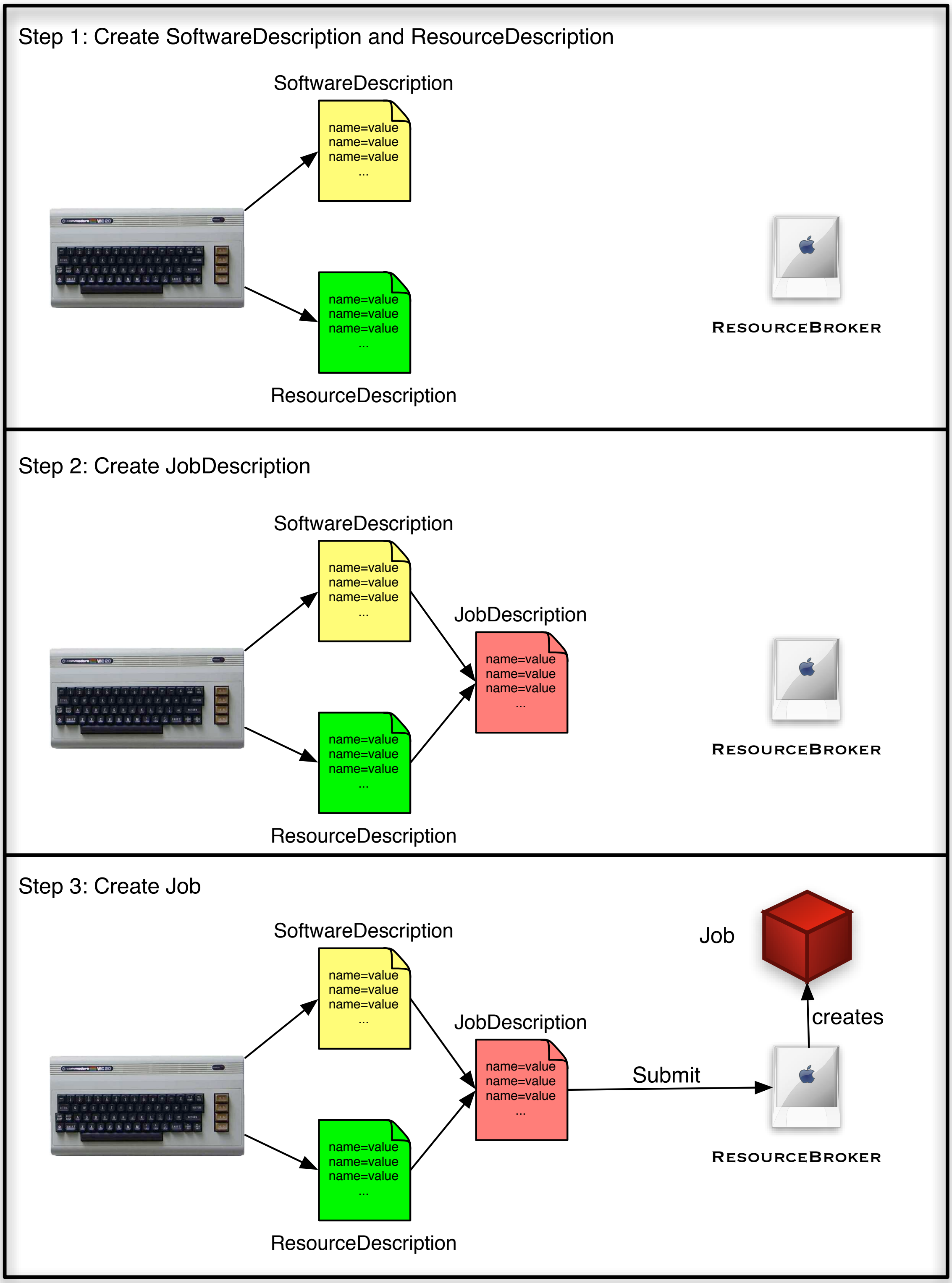

Figure 7: The steps in creating a job.

Checkpointing Jobs Before telling you how to "checkpoint" I'll take a step back and tell you what "checkpointing" is. Checkpointing is the process of saving the state of a job to a long term storage medium. In the future, at least the future according to Ray Kruzwell, people will be able to save the contents of their brains to hard drives, or whatever fancy thing takes the place of hard drives then. If you were to do so, then all you know, all you are, would be etched on these little spinning platers in one's and zero's. It would be like some type of insurance policy that would allow you to walk out the door and get hit by a car, but then be reconstituted from the little one's and zero's on this spinning platter. The process of writing all that is you to these hard drives is the process of checkpointing you. It would allow you to continue in the knowledge that if something were to go wrong, you could be restored to the you that existed just when you dumped your brain to disk.

Checkpointing a Job is easy, much easier then checkpointing you brain and much easier then stopping the crime-fighting onslaught that was "The Brown Hornet." One simply calls the `Checkpoint` function on a Job. All of the instrumentation involved with determining how to checkpoint the job is handled by the magic that is GAT.

Cloning Jobs GAT also allows for a job management primitive called "cloning." With the advent of Dolly, the now infamous ewe, the concept of cloning should be at least familiar to most. Cloning is the process of making a genetically identical organism from a single cell or individual by some asexual means. From one Dolly they made two, the immaculate conception. GAT also has a similar primitive. One can *clone* a job, make an exact copy of the running job. So, if your encephalon simulation has found a new home on a proper resource, such as the Earth Simulator, and you want to make an exact copy of the running job on your Commodore VIC-20 for old time sake, GAT allows you to do just that, though the utility of such in this case is at best dubious. To actually clone a Job one simply needs to call a single function `CloneJob` on a Job. That's it.

Migrating Jobs Migrating is a job primitive that can be thought of a being built up of two other primitives. Migration consists of first cloning a job, then stopping the original. Again, GAT makes migration easy in the extreme. Just call the `Migrate` function on a Job and you done.

2.2.8 Monitoring

Monitoring can be thought of a the process of spying. We can all remember the greatest of all secret agents, not that half-wit 007, but the one and only Lancelot Link Secret Chimp, staking out Dr. Strangemind, Dragon Woman, Ali Assa Seen, or The Baron, head of C.H.U.M.P. Lance would get one of his minions to embed sensors in the targets – home, office, car, or close to some other critical region. Lance would then wait a listen for any information that these sensors might find. He'd listen to hear what these monitorables might offer up in the way of information that would give him a tip as to their performance. Monitoring in GAT is no different, except, of course, for the fact that the monitorer and the monitoree aren't chimps.

Monitoring in GAT is the process of allowing one process, the listener, through various sensors embedded in a second process, a mointorable, to obtain information about this second process. Usually these embedded sensors are placed around critical regions in the code, for example at the end or start of the main loop of a program. These sensors allow the listener to listen for any information that these monitorables might offer up. The information that these sensors offer up is usually related to the performance of the mointorable, for example the number of iterations per second. Let take a look at some of the monitoring particulars.

Get Metrics Monitorable, a GAT abstraction tagging an instance as being able to be monitored, has a function called `GetMetrics`. This is the starting point for most monitoring. Upon calling this function the caller is returned a List of Metric instances. You say: "So, what is a Metric?" A Metric is an abstraction which represents a measurable quantity within the monitoring system. So, a call to the function `GetMetrics` on a Monitorable is a means for the caller to obtain from the Monitorable a description of the various quantities which it allows to be monitored. If only it were this easy for Lance.

Listening and not Listening for MetricEvents After getting from a Monitorable a List of Metric instances and deciding that you want to monitor one or the other of these metrics, what do you do then? Simply put, you register with the Monitorable to obtain notification if anything interesting happens. This is done by calling the function `AddMetricListener` on the Monitorable. Unlike most spam lists you can actually get off the notification list for a Monitorable. All one has to do to stop receiving notification of this Monitorable's state changes is to call the function `RemoveMetricListener` on the Monitorable.

2.3 Wait, There's More to GAT!

Beyond all the base GAT functionality I've described above, there's much more to GAT. If you look behind the curtain, you can be part of the prestidigitation that is GAT. If you are a "Grid" technology provider, you can help plug your technology in to the GAT framework. If you are a C developer, you can help maintain the C reference implementation of GAT. If you are an application developer, you can give much needed feedback on the GAT API. If you are a member of the hated underworld of C.H.U.M.P.[2],then, well, quite frankly we don't need your type around here; beat it.

[2] The sworn arch-enemies of A.P.E. "What's A.P.E.", you ask. A.P.E. is the **A**gency to **P**revent **E**vil, the umbrella organization under which Lancealot Link Secret Chimp and Mata Hairi operate.

3 GAT Object Model

3.1 C ain't Object Oriented

C ain't object oriented, but the GAT API, which exists in a language independent form and is "bound" to different languages, is objected oriented; hence, we have a problem. How does one express the object oriented constructs of the GAT API in an inherently non-object oriented language, C. The answer, with much gnashing of teeth and wringing of hands.

3.2 GAT Object Model

The GAT API is an object oriented API which makes use of several primitives of object oriented programming such as encapsulation, inheritance, interfaces, polymorphism, . . . C, however, doesn't naturally support any of these constructs. So, the introduction of these constructs in to the C version of GAT API in many cases required the reinvention of the wheel. However, you, the fearless application programmer, will never have to walk these minefields but will only reap the many benefits of this Sisyphean task.

The GAT object model consists of a set of "classes" arranged in a shallow "inheritance" tree, see figure 8.

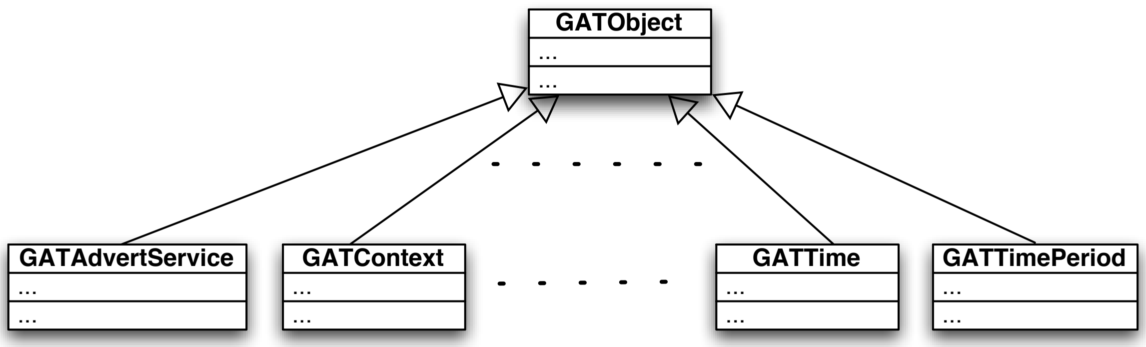

Figure 8: Structure of the GAT class inheritance tree.

Each GAT "class" consists of an opaque pointer, the details of which will never be of concern to the application programmer. Furthermore, as C does not natively support inheritance, the "inheritance" depicted in figure 8 is actually a stand-in for an inheritance with language support[3]. The means through which inheritance is actually implemented in GAT will never be of concern to the application programmer; however, the repercussions of this inheritance are of concern to the application programmer and are our next topic of discussion.

The class `GATObject`, from which all GAT classes inherit, posses a set of various functions, see figure 9.

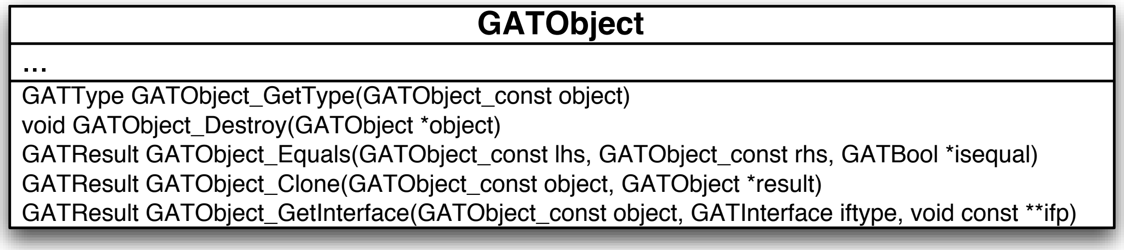

Figure 9: Structure of GATObject.

where `GATType` is an `enum` whose values correspond to the various GAT types, `GATObject_const` is a `const` version of `GATObject`, `GATResult` indicates a function's completion status, `GATBool` is a boolean, and `GATInterface` is and `enum` whose values correspond to the various GAT interfaces. (Interfaces are a topic which we will examine in the next subsection.) Each subclass of this `GATObject` possess a set of "corresponding" functions. So, for example, the class `GATTime` has the functions depicted in figure 10.

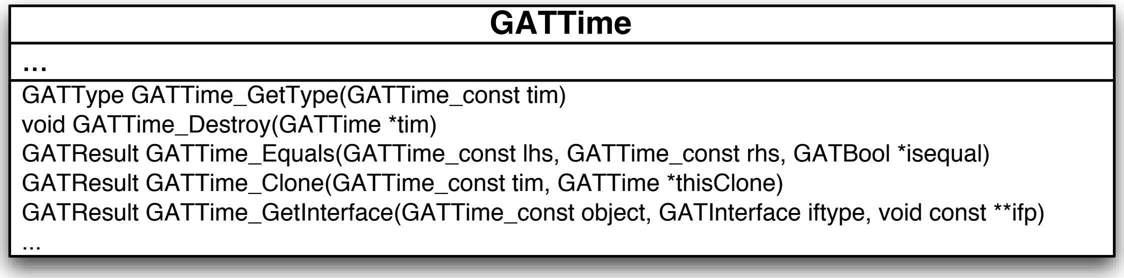

Figure 10: Structure of GATTime.

Furthermore, for each subclass of `GATObject` there exists a set of utility functions which allow one to convert this subclass to a `GATObject` and from a `GATObject`. So, for example, in the case of `GATTime` this set of utility functions is given by

[3] From this point on we will refer to the "inheritance" present in GAT and an inheritance with language support with the same term, inheritance. In addition we will refer to the various related terms "object," "superclass," "subclass,"... in GAT with the terms object, superclass, subclass,... with language support. No confusion should result from these slight abuses of terminology.

```
GATTime GATObject_ToGATTime(GATObject object)
GATObject GATTime_ToGATObject(GATTime derived)
GATTime_const GATObject_ToGATTime_const(GATObject_const object)
GATObject_const GATTime_ToGATObject_const(GATTime_const derived)
```

A class possessing these two properties, functions corresponding to the `GATObject` functions and the ability to convert to and from a `GATObject`, is said to inherit from `GATObject`. The full set of such GAT classes are depicted in figure 11.

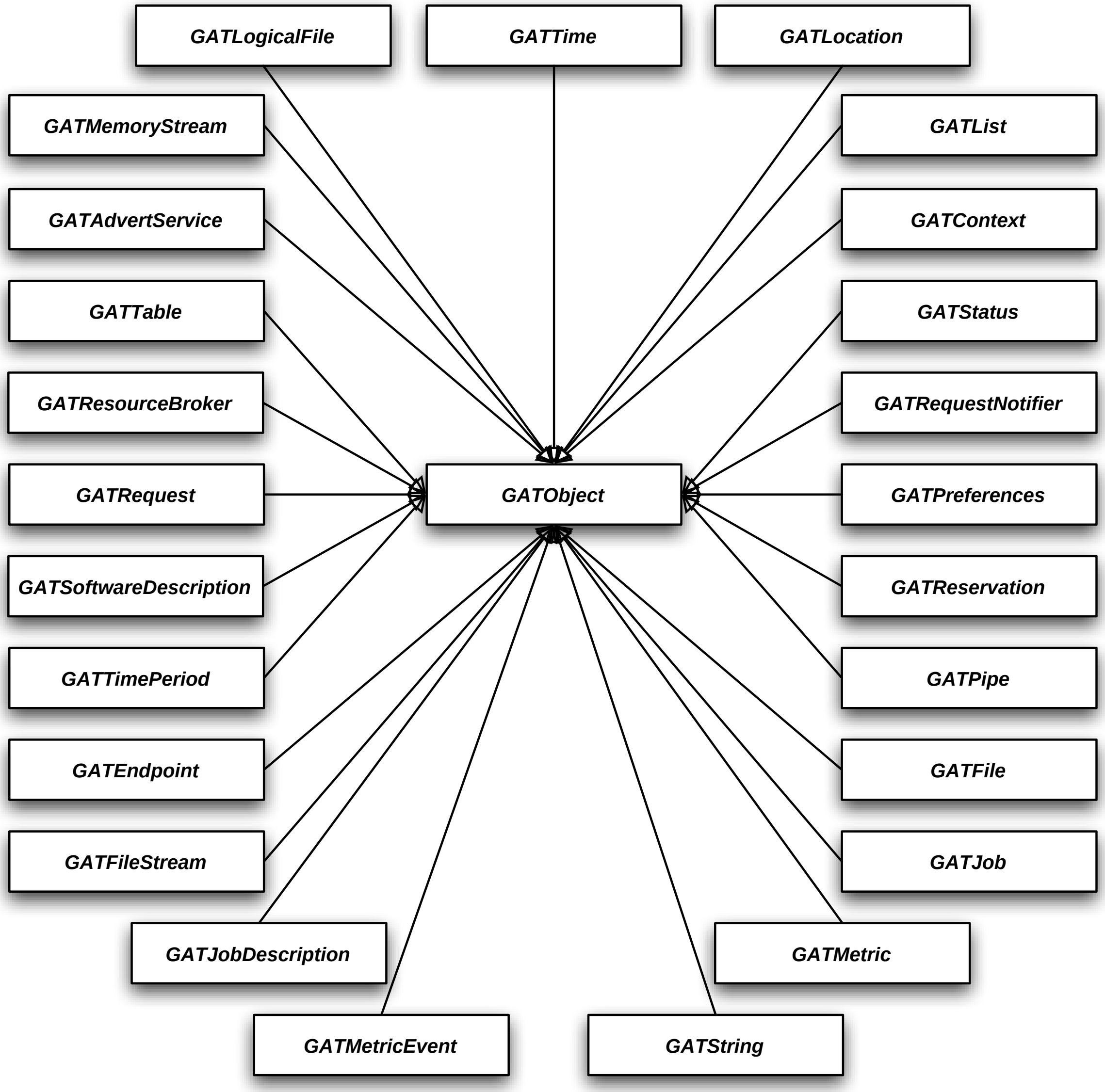

Figure 11: The full set of GAT classes.

3.3 GAT Interface Model

In addition to introducing notions of class, subclass, superclass, inheritance... GAT also introduces the notion of "interface." The GAT interface model consists of a set of "interfaces," see figure 12.

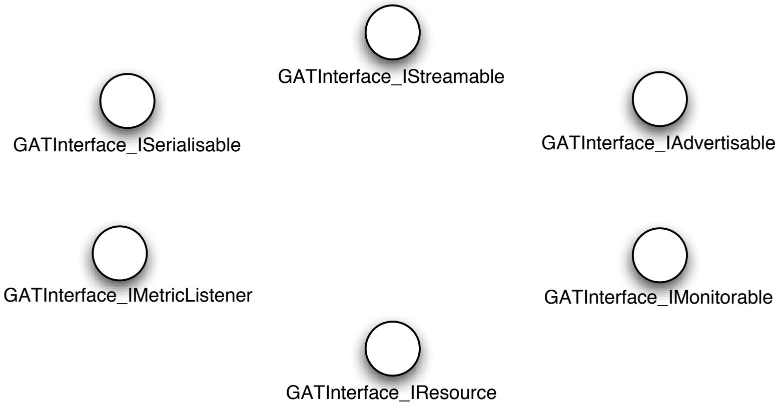

Figure 12: Structure of the GAT interface inheritance tree.

Each GAT "interface" consists of a `struct` containing function pointers. As C does not support interfaces, the "interfaces" depicted in figure 12 are simply a stand-in for interfaces with language support[4].

To clarify the concept of interfaces in GAT let us consider a few examples. The class `GATFile` realizes the interface `GATInterface_ISerialisable`, as rendered in figure 13.

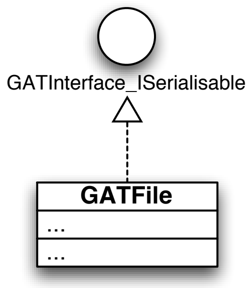

Figure 13: GATFile's realization of the interface GATInterface_ISerialisable.

As a result of this realization, there exists a set of `GATFile` functions implementing this realization. These functions are detailed in figure 14.

Similarly, the class `GATEndpoint` realizes the interface `GATInterface_ISerialisable`. Thus,

[4]From this point on we will refer to the "interfaces" present in GAT and interfaces with language support with the same term interfaces. No confusion should result from this slight abuse of terminology.

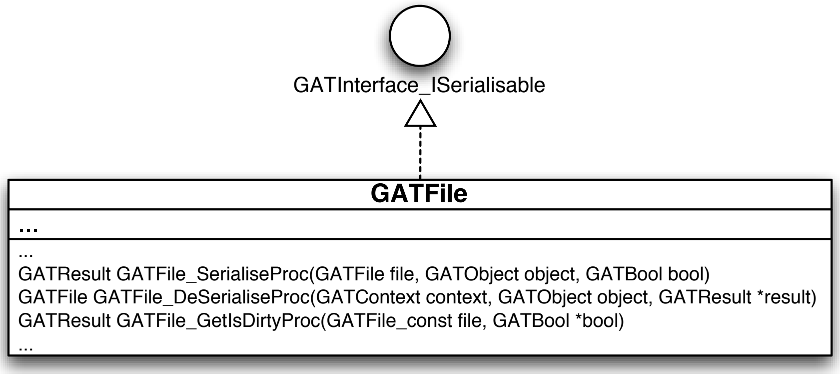

Figure 14: Details of GATFile's realization of the interface GATInterface_ISerialisable.

there exist a set of "corresponding" `GATEndpoint` functions implementing this realization, see figure 15.

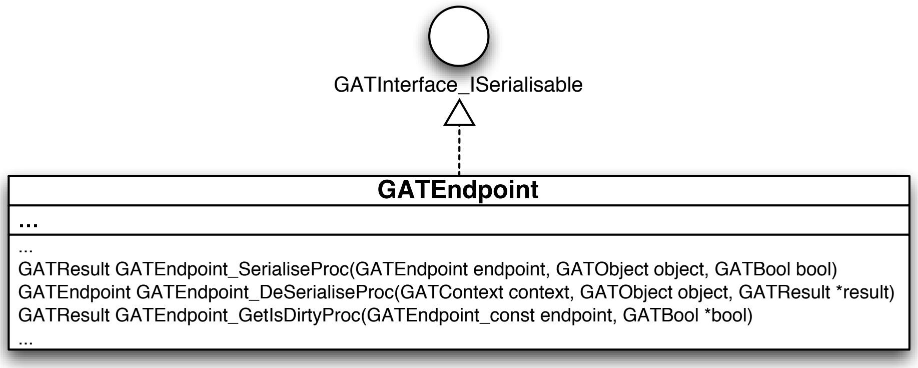

Figure 15: Details of GATEndpoint's realization of the interface GATInterface_ISerialisable.

As previously mentioned, an interface in GAT is a `struct` containing a set of function pointers. For example, these function pointers for the `GATInterface_ISerialisable` interface of `GATFile` are the functions specified in figure 14 above. To obtain reference to the `struct` containing these function pointers one must use the "GetInterface" member function of the apropos GAT object. As an example, in the case of a `GATFile` one would obtain such a `struct` as follows,

```
GATFile file;
GATResult result;
void const *serialisableInterface;

/* Create a new GATFile */
file = ...

/* Obtain the struct for the GATInterface_ISerialisable interface */
result = GATFile_GetInterface(file, GATInterface_ISerialisable, &serialisableInterface);
```

Upon successful return from the function `GATFile_GetInterface` the variable `*serialisable Interface` will point to a `struct` containing the aforementioned functions.

3.4 Some Not So Useful Programs

Each chapter of this part of the user's guide will contain in its final section a set of useful programs which illustrate the ideas covered earlier in the chapter. However, in this chapter we truthfully haven't covered enough ground to make anything of any real use; so, in this section we simply create some not so useful programs. Though, hopefully, they'll at least be educational. Here we go.

3.4.1 Getting an Object's Type

From the discussion earlier in this chapter, one can glean that each GAT class has a corresponding "GetType" function that returns a `GATType`, an `enum` whose values correspond to the various GAT types. (The full set of values which this `enum` may take on are detailed in Appendix A.) As our first example we will examine a program which obtains the `GATType` corresponding to a given GAT object. The full program is as follows

```
#include "GAT.h"

int main(void)
{
  GATType type;
  GATTime time;

  /* Create a GATTime corresponding to now */
  time = GATTime_Create(0);

  /* Obtain the GATType of the GATTime time */
  if( NULL != time )
  {
    type = GATTime_GetType(time);

    /* Destroy the GATTime time */
    GATTime_Destroy( &time );
  }

  return 0;
}
```

Let's examine this program line by line.

The first line in the program

```
#include "GAT.h"
```

is required of all GAT programs. This line includes the header `GAT.h` in which all the various functions and `struct`'s required by GAT are declared. Next the lines

```
int main(void)
{
  ...
}
```

are standard in any C program; thus, we won't belabor their details here. The next line

```
GATType type;
```

declares a variable `type` of type `GATType` which we will use to hold the GAT type of the GAT object we will study. The following line

```
GATTime time;
```

declares a variable `time` of type `GATTime`. As one can glean from figure 11, `GATTime` is a `GATObject`. Thus, as we mentioned previously, it has a "GetType" function. We will use this "GetType" function to assign the variable `type` the GAT type of `time`. The next lines in the program

```
/* Create a GATTime corresponding to now */
time = GATTime_Create(0);
```

create a `GATTime` instance. The function `GATTime_Create` has the following signature

```
GATTime GATTime_Create(GATdouble64 intime);
```

and is used to create instances of the class `GATTime`. In particular, it takes a `GATdouble64`, a GAT primitive type – the full set of GAT primitive types are detailed in the Appendix B – and returns a corresponding `GATTime` instance. The value passed to this function, when non-zero, is interpreted as the number of seconds elapsed since 00:00 hours, Jan 1, 1970 UTC, and the function returns a `GATTime` corresponding to this passed value. If the value 0 is passed to this function, then the returned `GATTime` corresponds to the number of seconds elapsed since 00:00 hours, Jan 1, 1970 UTC at which the function was called. In our case we pass 0 to the function; thus, the returned `GATTime` corresponds to the "instant" the function was called. Next the lines

```
/* Obtain the GATType of the GATTime time */
if( NULL != time )
{
  type = GATTime_GetType(time);

  ...
}
```

begin by first checking that the `GATTime time` is not `NULL`. A `NULL` instance may be returned from a "Create" statement if, for example, memory is running low and the program was unable to allocate sufficient memory to create a `GATTime` instance. Here we simple check this is not the case. After this check, we are guaranteed to have a valid `GATTime` instance `time` on which we may operate. The next line obtains the `GATType` corresponding to `time` and assigns this value to `type`. The full signature of the function `GATTime_GetType` which does this is as follows

```
GATType GATTime_GetType(GATTime_const time)
```

where, as one will recall, `GATTime_const` is a `const` version of `GATTime`. The next lines of the program

```
/* Destroy the GATTime time */
GATTime_Destroy( &time );
```

simply call the function

```
void GATTime_Destroy(GATTime *time)
```

This function deallocates any resources tied up by the `GATTime` instance `time` and should always be called when one is done with a `GATTime` instance. Similar "Destroy" functions exist for all GAT classes and should be called on the corresponding instances when one is done with such instances[5]. The final line of the program

```
return 0;
```

is part of standard C, and thus, we will not review it here.

3.4.2 Determining Object Equality

As discussed previously, each GAT class has a corresponding "Equals" function. In this subsection we will examine this "Equals" function.

First one may wonder what such an "Equals" function actually does as C is equipped with an extremely useful `==`. As `GATObject` instances are opaque pointers, one can use the `==` with which C is equipped to determine the equality of two such `GATObject` instances. For example, if we had two `GATTime` instances, `timeOne` and `timeTwo`, then we could determine their equality as follows

```
if( timeOne == timeTwo )
{
  printf("timeOne == timeTwo");
}
```

This would determine if the two opaque pointers `timeOne` and `timeTwo` point to the same region in memory, a useful piece of knowledge. However, it is often not enough.

There exist at least two notions of equivalence one commonly deals with: *pysical equivalence*, illustrated by the previous example using `==`, and *semantic equivalence*, which is what the "Equals" functions implement. For example, two `GATTime` instances are semantically equivalent if they currespond to the same number of seconds elapsed since 00:00 hours, Jan 1, 1970 UTC, the *epoch*. However, this semantic equivalence need not imply physical equivalence, which would only occur if the two instances are found to be equivalent using `==`.

The example for this section will hopefully clarify these concepts. The full example is as follows

[5]Note that one does not have to check that the argument to this function is NULL.

```
#include "GAT.h"
#include <stdio.h>

int main(void)
{
  GATBool isequal;
  GATTime timeOne;
  GATTime timeTwo;
  GATResult result;

  /* Create a GATTime corresponding to 1 hour after the epoch */
  timeOne = GATTime_Create( 3600 );

  /* Create a GATTime corresponding to 1 hour after the epoch */
  timeTwo = GATTime_Create( 3600 );

  /* Determine physical equivalence of timeOne and timeTwo */
  if( timeOne == timeTwo )
  {
    printf( "timeOne and timeTwo are physically equivalent\n" );
  }
  else
  {
    printf( "timeOne and timeTwo are not physically equivalent\n" );
  }

  /* Determine semantic equivalence of timeOne and timeTwo */
  if( (NULL != timeOne) && (NULL != timeTwo) )
  {
    result = GATTime_Equals( timeOne, timeTwo, &isequal );
    if( GAT_SUCCEEDED( result ) )
    {
      if( GATTrue == isequal )
      {
        printf( "timeOne and timeTwo are semantically equivalent\n" );
      }
      else
      {
        printf( "timeOne and timeTwo are not semantically equivalent\n" );
      }
    }
  }

  /* Destroy the GATTime timeOne */
  GATTime_Destroy( &timeOne );

  /* Destroy the GATTime timeTwo */
  GATTime_Destroy( &timeTwo );
```

```
  return 0;
}
```

Let us examine this program.

The lines

```
#include "GAT.h"
#include <stdio.h>

int main(void)
{
  ...
}
```

are now standard. They include the required GAT header `GAT.h`, the standard C header `stdio.h`, and the standard main function. The next lines

```
GATBool isequal;
GATTime timeOne;
GATTime timeTwo;
GATResult result;
```

declare the variable `isequal` of type `GATBool`, a type covered in Appendix B, the variable `timeOne` and `timeTwo`, each of type `GATTime`, and `result`, a variable of type `GATResult`. The type `GATResult` is simply `typedef`'d to be the GAT primitive type `GATint32`. Variables of type `GATResult` are used to return the completion status of a function. Various completion statuses correspond to various `GATResult` values. The various values and the semantics of the various values will not be covered here as that would take us a bit far afield; however, in Appendix C the various values and their semantics are covered in detail. The next lines

```
/* Create a GATTime corresponding to 1 hour after the epoch */
timeOne = GATTime_Create( 3600 );

/* Create a GATTime corresponding to 1 hour after the epoch */
timeTwo = GATTime_Create( 3600 );
```

create two `GATime` instances each corresponding to 3600 seconds after the epoch. The following lines

```
/* Determine physical equivalence of timeOne and timeTwo */
if( timeOne == timeTwo )
{
  printf( "timeOne and timeTwo are physically equivalent\n" );
}
else
{
  printf( "timeOne and timeTwo are not physically equivalent\n" );
}
```

determine if the two instances, `timeOne` and `timeTwo`, are physically equivalent and print the result of this determination. What will it print? The next lines

```
/* Determine semantic equivalence of timeOne and timeTwo */
if( (NULL != timeOne) && (NULL != timeTwo) )
{
  result = GATTime_Equals( timeOne, timeTwo, &isequal );
  if( GAT_SUCCEEDED( result ) )
  {
    if( GATTrue == isequal )
    {
      printf( "timeOne and timeTwo are semantically equivalent\n" );
    }
    else
    {
      printf( "timeOne and timeTwo are not semantically equivalent\n" );
    }
  }
}
```

first check that `timeOne` and `timeTwo` are both not `NULL`. They then call the function

```
GATResult GATTime_Equals( GATTime_const a, GATTime_const b, GATBool *isequal )
```

This function determines if the passed GATTime instances, `a` and `b`, are semantically equivalent. It returns the result of this determination in the `GATBool` pointed to by the passed `GATBool*`. The completion status of this function is returned through the return value of this function, a `GATResult`. The next lines

```
if( GAT_SUCCEEDED( result ) )
{
  ...
}
```

check that the call to the function `GATTime_Equals` completed sucessfully through use of the macro `GAT_SUCCEEDED`. Use of this macro is covered in Appendix C. The next lines

```
if( GATTrue == isequal )
{
  printf( "timeOne and timeTwo are semantically equivalent\n" );
}
else
{
  printf( "timeOne and timeTwo are not semantically equivalent\n" );
}
```

check the value of `isequal` against `GATTrue` and print out the result of this comparison. If `isequal` has value `GATTrue`, then `timeOne` and `timeTwo` are semantically equivalent. If `isequal` does not have value `GATTrue`, then `timeOne` and `timeTwo` aren't semantically equivalent. What will this print? The final lines of the program

```
/* Destroy the GATTime timeOne */
GATTime_Destroy( &timeOne );

/* Destroy the GATTime timeTwo */
GATTime_Destroy( &timeTwo );

return 0;
```

simply clean up the allocated `GATTime` instances and return 0 from the `main` function.

3.4.3 Cloning Objects

As mentioned previously, each GAT class has a "Clone" function. This subsection's example program will detail the use of such a "Clone" function.

The "Clone" function, with which every GAT class is equipted, allows an application programmer to make a deep clone of a GAT object. This is useful, for example, if one has an instance of a GAT class and wishes to make a copy, which is semantically equivalent to the original instance, to modify it in some way while keeping the original instance around un-modified. One could also conceive of a situation in which one would modify the original in some orthognal manner. The example of this section will show the use of such a "Clone" function.

The full example for this subsection is as follows

```
#include "GAT.h"
#include <stdio.h>

int main(void)
{
  GATBool isequal;
  GATTime timeOne;
  GATTime timeTwo;
  GATResult result;

  /* Create a GATTime corresponding to now */
  timeOne = GATTime_Create( 0 );

  /* Check timeOne is not NULL */
  if( NULL != timeOne )
  {
    /* Clone the GATTime timeOne */
    result =  GATTime_Clone( timeOne,  &timeTwo );

    /* Check success of call to GATTime_Clone */
    if( GAT_SUCCEEDED( result ) )
    {
      /* Determine if timeOne and timeTwo are semantically equivalent */
      result = GATTime_Equals( timeOne, timeTwo, &isequal );

      /* Check success of call to GATTime_Equals */
```

```
      if( GAT_SUCCEEDED( result ) )
      {
        /* Print result of call to GATTime_Equals */
        if( GATTrue == isequal )
        {
          printf( "timeOne and timeTwo are semantically equivalent\n" );
        }
        else
        {
          printf( "timeOne and timeTwo are not semantically equivalent\n" );
        }
      }
    }
  }

  /* Destroy the GATTime timeOne */
  GATTime_Destroy( &timeOne );

  /* Destroy the GATTime timeTwo */
  GATTime_Destroy( &timeTwo );

  return 0;
}
```

We have covered all the elements of this program previously, except one. The new element is the call to the function

```
GATResult GATTime_Clone(GATTime_const timeOne, GATTime * timeOneClone)
```

This function takes the passed `GATTime_const` instance `timeOne` and makes a deep clone of it. The deep clone is returned in the `GATTime` pointed to by the passed `GATTime*`. The completion status of this function is returned through the return value of this function, a `GATResult`.

3.4.4 Converting Objects

Earlier in this chapter we mentioned that for each GAT class there exists a set of companion functions which convert to and from instances of the particular GAT class and an instance of a `GATObject`. For example in the case of `GATTime` these functions are given by

```
GATTime GATObject_ToGATTime(GATObject object)
GATObject GATTime_ToGATObject(GATTime derived)
GATTime_const GATObject_ToGATTime_const(GATObject_const object)
GATObject_const GATTime_ToGATObject_const(GATTime_const derived)
```

The next example will be concerned with exercising these functions.

The full code for the next example is as follows

```
#include "GAT.h"

int main(void)
{
  GATTime time;
  GATObject object;

  /* Create a GATTime corresponding to now */
  time = GATTime_Create( 0 );

  /* Check time is not NULL */
  if( NULL != time )
  {
    /* Convert the GATTime to a GATObject */
    object = GATTime_ToGATObject( time );

    /* Convert a GATObject to a GATTime */
    time = GATObject_ToGATTime( object );
  }

 /* Destroy the GATTime time */
  GATTime_Destroy( &time );

  return 0;
}
```

Let us now examine this example program.

The first lines

```
#include "GAT.h"

int main(void)
{
  GATTime time;
  GATObject object;

  /* Create a GATTime corresponding to now */
  time = GATTime_Create( 0 );

  ...
}
```

are now standard; thus, we will not review them here. The next lines

```
/* Convert the GATTime to a GATObject */
object = GATTime_ToGATObject( time );

/* Convert a GATObject to a GATTime */
time = GATObject_ToGATTime( object );
```

contain some novel code. The first lines

```
/* Convert the GATTime to a GATObject */
object = GATTime_ToGATObject( time );
```

converts the `GATTime` instance `time` to a `GATObject` instance `object`. One should think of this conversion as something akin to a cast. In particular, no new memory or resources are allocated in the processs of this conversion and the returned `GATObject` refers to the same allocated object as the original `GATTime`. Thus, one need not call the "Destroy" function on the resulting `GATObject` *and* the associated `GATTime`. One need only call the "Destroy" function on the `GATTime` *or* the `GATObject` instance. The next lines

```
/* Convert a GATObject to a GATTime */
time = GATObject_ToGATTime( object );
```

convert back from the `GATObject` instance `object` to a `GATTime` instance `time`. The remainder of the program

```
/* Destroy the GATTime time */
GATTime_Destroy( &time );

return 0;
```

is now standard; hence, we will not cover these lines in detail.

3.4.5 Using an Object's Interface

Normally the use of an interface is rather messy, pointers to pointers to pointers to `struct`'s full of function pointers. So, GAT introduces various utility functions which make the use of a given interface easy. These various functions are grouped according to the interface they support. So, for most interfaces in figure 12 there exists a grouping of utility functions which facilitate the use of the given interface. A given group of these various utility functions can loosely be thought of as representing the corresponding interface. So, loosely speaking, we can represent the interfaces in GAT as portrayed in figure 16.

In this subsection we will focus on an example which uses these utility functions. In particular the example will focus on using the grouping of utility functions associated with the `GATInterface_ISerialisable` interface of a `GATTime` instance. Through the use of these utility functions we will exercise the interface `GATInterface_ISerialisable` to serialise a `GATTime` instance, an operation that will be extremely useful in future.

The full code for this section's example is as follows

```
#include "GAT.h"
#include <stdio.h>

int main(void)
{
  void *buffer;
```

<<interface>>
GATInterface_ISerialisable

GATResult GATSerialisable_Serialise(GATObject object, GATObject stream, GATBool clearDirty)
GATObject GATSerialisable_DeSerialise(GATContext context, GATObject stream, GATResult *result)
GATResult GATSerialisable_GetIsDirty(GATObject_const object, GATBool *isdirty)

<<interface>>
GATInterface_IStreamable

GATResult GATStreamable_Read(GATObject object, void *buffer, GATuint32 size, GATuint32 *readBytes)
GATResult GATStreamable_Write(GATObject object, void const *buffer, GATuint32 size, GATuint32 *writtenBytes)
GATResult GATStreamable_Seek(GATObject object, GATOrigin origin, GATint32 offset, GATuint32 *new_position)

<<interface>>
GATInterface_IResource

GATResult GATResource_GetResourceDescription(GATObject_const object, GATResourceDescription_const *description)
GATResult GATResource_GetReservation(GATObject_const object, GATReservation_const *reservation)

<<interface>>
GATInterface_IMonitorable

GATResult GATMonitorable_AddMetricListener(GATObject object, GATMetricListener listener, void *listener_data, GATMetric metric, GATuint32 *cookie)
GATResult GATMonitorable_RegisterPolling(GATObject object, GATMetric metric, GATMetricEvent *event, GATuint32 *cookie)
GATResult GATMonitorable_RemoveRegisteredMetric(GATObject object, GATMetric metric, GATuint32 cookie)
GATResult GATMonitorable_GetMetrics(GATObject_const object, struct GATList_GATMetric_handle **metrics)

Figure 16: The various GAT interfaces and the associated utility functions.

```
GATTime time;
GATResult result;
GATuint32 counter;
GATuint32 bufferSize;
GATObject timeObject;
GATObject streamObject;
GATMemoryStream stream;

/* Create a GATTime corresponding to now */
time = GATTime_Create( 0 );

/* Create a GATMemoryStream */
stream = GATMemoryStream_Create(0, 0, GATFalse);

/* Check time and stream are not NULL */
if( (NULL != time) && (NULL != stream) )
{
  /* Convert the GATTime to a GATObject */
  timeObject = GATTime_ToGATObject(time);

  /* Convert the GATMemoryStream to a GATObject */
  streamObject = GATMemoryStream_ToGATObject(stream);
```

```
    /* Serialize timeObject to streamObject */
    result = GATSerialisable_Serialise( timeObject, streamObject, GATFalse );

    /* Check for serialization success */
    if( GAT_SUCCEEDED( result ) )
    {
      /* Obtain the buffer with the serialized GATTime */
      buffer = GATMemoryStream_GetBuffer( stream, &bufferSize, GATFalse );

      /* Print out GATTime serialization */
      for( counter = 0; counter < bufferSize; counter++ )
      {
        printf( "The next four bytes of now are %x\n", ( GATuint32[] buffer )[counter] );
      }
    }
  }

 /* Destroy the GATTime */
  GATTime_Destroy( &time );

 /* Destroy the GATMemoryStream */
 GATMemoryStream_Destroy( &stream );

 return 0;
}
```

Let us examine this example.

The first lines of the example are now standard

```
#include "GAT.h"
#include <stdio.h>

int main(void)
{
  void *buffer;
  GATTime time;
  GATResult result;
  GATuint32 counter;
  GATuint32 bufferSize;
  GATObject timeObject;
  GATObject streamObject;
  GATMemoryStream stream;

  /* Create a GATTime corresponding to now */
  time = GATTime_Create( 0 );
  ...
}
```

The only novel part of this code is the introduction of `GATMemoryStream`. A `GATMemoryStream` is an internal class used by GAT whose details are not of concern to our current train of thought beyond the fact that an instance of a GATMemoryStream can be thought of as a buffer which can be used to hold our serialized GATTime. The next lines

```
/* Create a GATMemoryStream */
stream = GATMemoryStream_Create(0, 0, GATFalse);

/* Check time and stream are not NULL */
if( (NULL != time) && (NULL != stream) )
{
  ...
}
```

create a GATMemoryStream then proceed only if the created GATTime and GATMemoryStream are not NULL. The following lines

```
/* Convert the GATTime to a GATObject */
timeObject = GATTime_ToGATObject(time);

/* Convert the GATMemoryStream to a GATObject */
streamObject = GATMemoryStream_ToGATObject(stream);
```

use the various conversion utility functions to convert the GATTime and GATMemoryStream to GATObject's. The proceeding lines

```
/* Serialize timeObject to streamObject */
result = GATSerialisable_Serialise( timeObject, streamObject, GATFalse );

/* Check for serialization success */
if( GAT_SUCCEEDED( result ) )
{
  ...
}
```

contain the novel code of this example. These lines employ the utility function

```
GATResult GATSerialisable_Serialise(GATObject o, GATObject s, GATBool c)
```

which in turn uses the interface GATInterface_ISerialisable, to serialize the passed GATObject instance timeObject to the passed GATObject instance streamObject. The final argument to this function is a GATBool which indicates if the "dirty" flag of the GATObject instance timeObject should be cleared upon serialization. For example, this "dirty" flag can be used to keep track of the possible difference between a serialized version of a particular instance and the in-memory version of that same instance. In our example we don't care about such distinctions. In addition we should note that the first argument to this function must be a GATObject realizing the GATInterface_ISerialisable interface and the second argument to this function must be a GATObject realizing the GATInterface_IStreamable interface. The very next lines simply check to see that this serialization completed successfully. The following lines

```
/* Obtain the buffer with the serialized GATTime */
buffer = GATMemoryStream_GetBuffer( stream, &bufferSize, GATFalse );

/* Print out GATTime serialization */
for( counter = 0; counter < bufferSize; counter++ )
{
  printf( "The next four bytes of now are %x\n", ( GATuint32[] buffer )[counter] );
}
```

first obtain the buffer holding the serialized version of the `GATTime`, the buffer is an array of `GATuint32`'s, then simply proceed to print out each `GATuint32` in this buffer. The final lines of the example

```
/* Destroy the GATTime */
GATTime_Destroy( &time );

/* Destroy the GATMemoryStream */
GATMemoryStream_Destroy( &stream );

return 0;
```

are now standard; thus, we will not examine them in detail.

4 File Management

4.1 Files, Folders, and the Letter "F"

Everything is a file! Source code, a file. Compiled code, a file. Shared object, a file... A file is the central metaphor which lies at the heart of most, if not all, modern computer systems. So, if GAT is at all worth its salt, then it should allow the application programmer to wangle files to their heart's content. In fact it does this and more, providing lasso and lessons in file bulldogging to the aspiring GAT cowboy.

4.2 The File Package

The file package is relatively simple in its content. It only contains a single class `GATFile`. However, don't let this simplicity fool you; this one class is packed to the brim with extra goodness. It lets you, the application programmer, marionette files to your hearts content. Creating, deleting, copying, moving... any file anywhere. Lets see how it's done.

4.2.1 Creating and Destroying File Instances

Before we can wrangle a `GATFile`, we first need to create a `GATFile`, and before we create a `GATFile` we need to learn how to create instances of a two other classes required in the creation of a `GATFile`. In particular we need to understand how to create an instance of a `GATLocation`, and a `GATContext`. Lets get crackin' with these preliminaries.

Specifying the location of a file is accomplished through the use of the auxiliary class named, appropriately enough, `GATLocation`. So, taking one step backwards before taking one step forward we'll consider how to create a `GATLocation`.

A `GATLocation` is created using the single call `GATLocation_Create`. This signature of this function is as follows

```
GATLocation GATLocation_Create(const char *uri)
```

The `uri` argument to this function is an `char *` representation of a URI. This function returns a `NULL`, in the case of an error, or a `GATLocation` corresponding to the passed URI. After creating a `GATLocation` and bending it to our will, we will need to destroy the `GATLocation`. A simple call to the function

```
void GATLocation_Destroy(GATLocation *loc)
```

destroys the passed `GATLocation` and frees up any resources it may have held.

The second class we need to learn how to create instances of is a `GATContext`. An instance of a `GATContext` is used to store various state information related to GAT calls, such as security information or the errors in the current call stack. A `GATContext` instance is created using the following function

```
GATContext GATContext_Create(void)
```

This function returns a `GATContext` or `NULL` upon error. Upon making such a created `GATContext` jump through the proper hoops, we will need to put it out to pasture. This is done through a call to the function

```
void GATContext_Destroy(GATContext *ctx)
```

which destroys the passed `GATContext` instance and frees up any resources held by the passed `GATContext`.

Now that we've got these preliminaries under control lets see how to use such to create a `GATFile`. A `GATFile` instance is created through a single call to the following function

```
GATFile
  GATFile_Create( GATContext context,
                             GATLocation_const location,
                             GATPreferences_const preferences )
```

The first argument to this function is a `GATContext` instance used to store state information related to the various calls of the created `GATFile`. The second argument is a `GATLocation` instance specifying the location which the created `GATFile` is to correspond. The final argument is a `GATPreferences`, in most of our examples this final argument will be `NULL`, the class `GATPreferences` is covered in detail in Appendix D. The return value of this function will be a `GATFile` instance with the passed location or `NULL`, on error.

Now that we know how to create a `GATFile` instance we should learn how to destroy such an instance as we wouldn't want to leave any resources hanging about. This is done through a call to the function

```
void GATFile_Destroy(GATFile *file)
```

This function destroys the passed `GATFile` instance and frees up any resources this instance might have tied up.

4.2.2 Copying, Moving, and Deleting File Instances

As we've now the experience of creating and deleting a `GATFile` instance corresponding to a physical file at a particular location, we can next move on to some file manipulation routines. The first trinity which we we explore in this realm is that of copying, moving, and deleting a file.

Let us first examine the process of copying a `GATFile`. Upon creating a `GATFile` instance one can easily copy such a `GATFile` instance to a new location using the single call

```
GATResult GATFile_Copy( GATFile_const file,
                                      GATLocation_const targetLocation,
                                      GATFileMode mode )
```

The first argument to this function is the `GATFile` instance one wishes to copy. The second argument is a `GATLocation` instance indicating to where one wishes to copy the `GATFile` instance. The final argument to this function is a `GATFileMode`. `GATFileMode` is an enumeration indicating how the copy operation should behave if there exists a physical file in the selected destination location. The possible values that a `GATFileMode` may take on are as follows

GATFileMode	**Resultant function semantics**
GATFileMode_FailIfExists	Fail if there exists a file at the destination location.
GATFileMode_Overwrite	Overwrite any file, if extant, at the destination location.

Table 1: The full set of GATFileMode values.

The return value of this function is a `GATResult`, covered in appendix C.

For example, if one has a `GATFile` instance `file` that one wishes to copy to a location described by the `GATLocation` instance `targetLocation` such that if a destination file exists it is overwritten, then the call would look at follows

```
GATFile file;
GATResult result;
GATLocation targetLocation;

file = ...
targetLocation = ...

result = GATFile_Copy( file, targetLocation, GATFileMode_Overwrite );
```

Similarly, if one has a `GATFile` instance `file` that one wishes to copy to a location described by the `GATLocation` instance `targetLocation` such that the copy call fails if there exists a destination file, then the call would look at follows

```
GATFile file;
GATResult result;
GATLocation targetLocation;

file = ...
targetLocation = ...

result = GATFile_Copy( file, targetLocation, GATFileMode_FailIfExists );
```

Lets next move on the the process of moving a `GATFile`. One moves a `GATFile` instance through the call

```
GATResult GATFile_Move( GATFile_const file,
                                       GATLocation_const targetLocation,
                                       GATFileMode mode)
```

The first argument to this function is the `GATFile` instance one wishes to move. The second argument is a `GATLocation` instance indicating where one wishes to move the `GATFile` instance. The final argument is a `GATFileMode` indicating how this function call should behave if there exists a file in the selected destination location. The return value of this function is a `GATResult`, covered in appendix C.

As an example, let us consider a `GATFile` instance `file` that one wishes to move to a location described by the `GATLocation` instance `targetLocation` such that the move call fails if there exists a destination file. The call would look at follows

```
GATFile file;
GATResult result;
GATLocation targetLocation;

file = ...
targetLocation = ...

result = GATFile_Move( file, targetLocation, GATFileMode_Overwrite );
```

If one did not wish to overwrite the destination file, is extant, then one would pass as the final argument in this function the constant `GATFileMode_FailIfExists`.

Finally let us examine the process of deleting a physical file corresponding to a `GATFile` instance. This is accomplished using the following function call

```
GATResult GATFile_Delete(GATFile_const file)
```

The first and only argument to this function is the `GATFile` instance representing the physical file that one wishes to delete. The return value, unsurprisingly, is a `GATResult`, covered in appendix C.

As an example of this call in the wild, consider a `GATFile` instance `file` corresponding to a physical file that one wishes to delete. Using the above call to delete this physical file, one would use code of the following form

```
GATFile file;
GATResult result;

file = ...

result = GATFile_Delete( file );
```

Simple, no?

4.2.3 Examining File Instances

In addition to the relatively pedestrian tasks of copying, moving and deleting a file, GAT allows for one to determine various properties of a file through one simple interface. In particular, GAT allows for one to determine if the file is readable or writable. Also, GAT allows for one to obtain the length of a file in bytes, a `GATTime` indicating the last write time of a file, or a `GATLocation` indicating the location of a file. Lets take a look at how to root about in the internals of a `GATFile`.

First lets look at how to determine if we can read a file. This is accomplished through a call to the function

```
GATResult GATFile_IsReadable(GATFile_const file)
```

The passed `GATFile` instance is the file one wishes to examine for readability. If this passed file is readable, then this function returns a `GATResult` of `GAT_SUCCESS`. If the passed file is not readable, then this function returns a value equal to `GAT_FALSE`. The signature for this function is a bit perverse. A cleaner signature for this function would have been the following

```
GATResult GATFile_IsReadable(GATFile_const file, GATBool *isReadable)
```

But hey, no one is perfect.

To determine if a file is writable one plays a similar game using the function

```
GATResult GATFile_IsWritable(GATFile_const file)
```

The passed `GATFile` instance is the file one wishes to examine for writability. If this passed file is writable, then this function returns a `GATResult` of `GAT_SUCCESS`. If the passed file is not writable, then this function returns a value equal to `GAT_FALSE`. Again, GAT sticks to this perverse, if not a bit confusing, function signature.

To see how these functions work, lets take a look at a call to determine if a `GATFile` instance `file` is readable. A code snippet for such a task is as follows

```
GATFile file;
GATResult result;

file = ...

result = GATFile_IsReadable( file );
if( GAT_SUCCESS == result )
{
  /* File is readable, you can read it here */
  ...
}

if( GAT_FALSE == result )
{
  /* File is not readable, you can not read it here */
}
```

In addition to determining if a file is readable or writable one can determine other properties of the physical file corresponding to a `GATFile` instance. For example one can determine the length of the physical file corresponding to a `GATFile` instance using the function

```
GATResult GATFile_GetLength(GATFile_const file, unsigned long *length)
```

The passed `GATFile` instance corresponds to the file one wishes to examine the length of and upon successful completion of this function the `unsigned long` returns the number of bytes in the physical file corresponding to the passed `GATFile` instance. Finally, the returned `GATResult` corresponds to the completion status of this function, `GATResult` is covered in Appendix C.

So, for example, to obtain the length of a `GATFile` instance one would proceed as in this code snippet

```
GATFile file;
GATResult result;
unsigned long length;

file = ...

result = GATFile_GetLength( file, &length );

if( GAT_SUCCEEDED( result ) )
{
  /* File is length bytes long */
}
```

Similarly, we can determine the most recent time at which the physical file corresponding to a `GATFile` instance was written to using the function

```
GATResult GATFile_LastWriteTime(GATFile_const file, GATTime *lw_time)
```

The passed `GATFile` instance is the file one wishes to examine the last write time of and upon successful completion of this function the `GATTime` returns the last write time of the physical file corresponding to the passed `GATFile` instance. Finally, the returned `GATResult` corresponds to the comletion status of this function, `GATResult` is covered in Appendix C.

As an example of using this function in practice we can take a look at this code snippet which obtains the last write time of the `GATFile` instance `file`

```
GATFile file;
GATResult result;
GATTime lastWriteTime;

file = ...

result = GATFile_LastWriteTime( file, &lastWriteTime );

if( GAT_SUCCEEDED( result ) )
{
  /* File was last written at lastWriteTime */
}
```

Finally, one can obtain the location of a physical file corresponding to a GATFile instance using the following function

```
GATLocation_const GATFile_GetLocation(GATFile_const file)
```

The passed GATFile instance is the file one wishes to examine the location of and the returned GATLocation is the location of the passed GATFile instance.

For example, to obtain the GATLocation instance corresponding to a GATFile instance file one would proceed as follows

```
GATFile file;
GATLocation_const location;

file = ...

location = GATFile_GetLocation( file );
```

4.3 Some Useful Programs

As we've now covered the basics of the file management package, we can move on to create some useful programs. Of the most common tools that are used on a daily basis in unix operating systems are cp, mv, rm, and ls. However, there exist no common "grid enabled" versions of these work-horses of the UNIX world. In this section we will create gird versions of these common tools.

4.3.1 A Fancy-Pants cp

The program cp is one of the most useful, yet extremely simple, programs that the fingers of a Unix user has access to. In its most simple form, it is used to copy a file to a new location. For example, to copy the file source to the location destination the program cp could be used as follows

```
% cp source destination
```

Here we will create a "grid enabled" version of this bread and butter of the Unix world. The full program is as follows

```
#include <stdio.h>
#include "GAT.h"

int main( int argc, char *argv[] )
{
  GATResult result;
  GATFile sourceFile;
  GATContext context;
  GATLocation sourceLocation;
  GATLocation destinationLocation;

  /* Check command line syntax */
  if( 3 != argc )
```

```
{
   printf("usage: %s source destination\n", argv[0]);

   return 1;
}

/* Set result to a memory failure */
result = GAT_MEMORYFAILURE;

/* Create GATLocation sourceLocation */
sourceLocation = GATLocation_Create( argv[1] );

/* Check previous GATLocation creation */
if( NULL != sourceLocation )
{
  /* Create GATLocation destinationLocation */
  destinationLocation = GATLocation_Create( argv[2] );

  /* Check previous GATLocation creation */
  if( NULL != destinationLocation )
  {
    /* Create GATContext context */
    context = GATContext_Create();

    /* Check previous GATContext creation */
    if( NULL != context )
    {
      /* Create GATFile sourceFile */
      sourceFile = GATFile_Create( context, sourceLocation, NULL );

      /* Check GATFile creation */
      if( NULL != sourceFile )
      {
        /* Copy sourceFile to destinationLocation */
        result = GATFile_Copy( sourceFile, destinationLocation, GATFileMode_Overwrite );

        /* Destroy GATFile sourceFile */
        GATFile_Destroy( &sourceFile );
      }

      /* Destroy GATContext context */
      GATContext_Destroy( &context );
    }

    /* Destroy GATLocation destinationLocation */
    GATLocation_Destroy( &destinationLocation );
  }

  /* Destroy GATLocation sourceLocation */
```

```
    GATLocation_Destroy( &sourceLocation );
  }

  /* Check result for success and print error */
  if( GAT_FAILED( result) )
  {
    printf( "An error has occurred during the copy operation\n");

    return 1;
  }

  return 0;
}
```

This entire program contains no novel code, all of its functions have been previously examined; so, we will not examine it line by line.

4.3.2 Wow Your Friends with mv

Another program which is manna to Unix users everywhere is the program `mv`. The program `mv` moves a specified file to a specified location. So, for example, to move the file `source` to the location `destination` the program `mv` would be used as follows

```
% mv source destination
```

The next program will be a "grid enabled" version of this common command line tool. The full program is as follows

```
#include <stdio.h>
#include "GAT.h"

int main( int argc, char *argv[] )
{
  GATResult result;
  GATFile sourceFile;
  GATContext context;
  GATLocation sourceLocation;
  GATLocation destinationLocation;

  /* Check command line syntax */
  if( 3 != argc )
  {
     printf("usage: %s source destination\n", argv[0]);

     return 1;
  }

  /* Set result to a memory failure */
  result = GAT_MEMORYFAILURE;
```

```
/* Create GATLocation sourceLocation */
sourceLocation = GATLocation_Create( argv[1] );

/* Check previous GATLocation creation */
if( NULL != sourceLocation )
{
  /* Create GATLocation destinationLocation */
  destinationLocation = GATLocation_Create( argv[2] );

  /* Check previous GATLocation creation */
  if( NULL != destinationLocation )
  {
    /* Create GATContext context */
    context = GATContext_Create();

    /* Check previous GATContext creation */
    if( NULL != context )
    {
      /* Create GATFile sourceFile */
      sourceFile = GATFile_Create( context, sourceLocation, NULL );

      /* Check GATFile creation */
      if( NULL != sourceFile )
      {
        /* Move sourceFile to destinationLocation */
        result = GATFile_Move( sourceFile, destinationLocation, GATFileMode_Overwrite );

        /* Destroy GATFile sourceFile */
        GATFile_Destroy( &sourceFile );
      }

      /* Destroy GATContext context */
      GATContext_Destroy( &context );
    }

    /* Destroy GATLocation destinationLocation */
    GATLocation_Destroy( &destinationLocation );
  }

  /* Destroy GATLocation sourceLocation */
  GATLocation_Destroy( &sourceLocation );
}

/* Check result for success and print error */
if( GAT_FAILED( result) )
{
  printf( "An error has occurred during the move operation\n");
```

```
    return 1;
  }

  return 0;
}
```

Again this entire program contains no novel code, all of its functions have been previously examined; so, we will not examine it line by line.

4.3.3 From rm to RM

When you end up running out of hard drive space,"Of course I haven't downloaded avi versions of all the Star Trek episodes!", one of the most useful programs to have is the Unix program `rm`. The program `rm` is used to remove files. So, for example, to remove the file `file` from the hard drive `rm` would be used as follows

```
% rm file
```

Our next program will be, in line with our previous creations, a "grid enabled" `rm`. The full program is here

```
#include <stdio.h>
#include "GAT.h"

int main( int argc, char *argv[] )
{
  GATResult result;
  GATFile sourceFile;
  GATContext context;
  GATLocation sourceLocation;

  /* Check command line syntax */
  if( 2 != argc )
  {
     printf("usage: %s file\n", argv[0]);

     return 1;
  }

  /* Set result to a memory failure */
  result = GAT_MEMORYFAILURE;

  /* Create GATLocation sourceLocation */
  sourceLocation = GATLocation_Create( argv[1] );

  /* Check previous GATLocation creation */
  if( NULL != sourceLocation )
  {
      /* Create GATContext context */
```

```
      context = GATContext_Create();

      /* Check previous GATContext creation */
      if( NULL != context )
      {
        /* Create GATFile sourceFile */
        sourceFile = GATFile_Create( context, sourceLocation, NULL );

        /* Check GATFile creation */
        if( NULL != sourceFile )
        {
          /* Delete sourceFile */
          result = GATFile_Delete( sourceFile );

          /* Destroy GATFile sourceFile */
          GATFile_Destroy( &sourceFile );
        }

        /* Destroy GATContext context */
        GATContext_Destroy( &context );
      }

    /* Destroy GATLocation sourceLocation */
    GATLocation_Destroy( &sourceLocation );
  }

  /* Check result for success and print error */
  if( GAT_FAILED( result) )
  {
    printf( "An error has occurred during the delete operation\n");

    return 1;
  }

  return 0;
}
```

This entire program contains no novel code; so, we will not examine it line by line.

4.3.4 A ls -l of sorts

Another common Unix command line tool is `ls`. The program `ls` lists the contents of a given directory. In addition the program `ls` can be used to examine various file properties. For example the following command line call to the program `ls` used with the `-l` option

```
% ls -l GATErrors.h
```

will print out the following information about the file `GATErrors.h`

```
-rw-r--r--  1 leonardo  masters  14477 15 Apr 11:13 GATErrors.h
```

The first bit of this printout `-rw-r--r--` indicates who can read and write the file `GATErrors.h`; the portion `14477` indicates the length of the file `GATErrors.h` in bytes, and the portion `15 Apr 11:13` indicates the last write time of the file `GATErrors.h`. As has become our habit, we will create a "grid enabled" version of `ls -l`. The full program is as follows

```
#include <stdio.h>
#include "GAT.h"

int main( int argc, char *argv[] )
{
  char writeChar;
  char readChar;
  GATResult result;
  GATFile sourceFile;
  GATContext context;
  unsigned long length;
  GATTime lastWriteTime;
  GATLocation sourceLocation;

  /* Check command line syntax */
  if( 2 != argc )
  {
     printf("usage: %s file\n", argv[0]);

     return 1;
  }

  /* Set result to a memory failure */
  result = GAT_MEMORYFAILURE;

  /* Create GATLocation sourceLocation */
  sourceLocation = GATLocation_Create( argv[1] );

  /* Check previous GATLocation creation */
  if( NULL != sourceLocation )
  {
      /* Create GATContext context */
      context = GATContext_Create();

      /* Check previous GATContext creation */
      if( NULL != context )
      {
        /* Create GATFile sourceFile */
        sourceFile = GATFile_Create( context, sourceLocation, NULL );

        /* Check GATFile creation */
        if( NULL != sourceFile )
```

```
{
  /* Obtain  GATFile length */
  result = GATFile_GetLength( sourceFile,  &length );

  /* Check call to GATFile_GetLength
  if( GAT_SUCCEEDED( result ) )
  {
    /* Obtain GATFile lastWriteTime */
    result = GATFile_LastWriteTime( sourceFile, &lastWriteTime );

    /* Check call to GATFile_LastWriteTime */
    if( GAT_SUCCEEDED( result ) )
    {
      /* Determine readability */
      result = GATFile_IsReadable( sourceFile );

      /* Check call to GATFile_IsReadable */
      if( GAT_SUCCEEDED( result ) )
      {
        /* Set readChar */
        readChar = '-';
        if( GAT_SUCCESS == result )
        {
          readChar = 'r';
        }

        /* Determine writability */
        result = GATFile_IsWritable( sourceFile );

        /* Check call to GATFile_IsWritable */
        if( GAT_SUCCEEDED( result ) )
        {
          /* Set writeChar */
          writeChar = '-';
          if( GAT_SUCCESS == result )
          {
            writeChar = 'w';
          }

          /* Print results in the form: rw length lastWriteTime file */
          printf( "%c%c %d %d %s\n",
                  readChar, writeChar, length,
                  GATTime_GetTime(lastWriteTime), argv[1] );
        }
      }

      /* Destroy GATTime */
      GATTime_Destroy( &lastWriteTime );
    }
```

```
          }

          /* Destroy GATFile sourceFile */
          GATFile_Destroy( &sourceFile );
        }

        /* Destroy GATContext context */
        GATContext_Destroy( &context );
      }

    /* Destroy GATLocation sourceLocation */
    GATLocation_Destroy( &sourceLocation );
  }

  /* Check result for success and print error */
  if( GAT_FAILED( result) )
  {
    printf( "An error has occurred during the delete operation\n");

    return 1;
  }

  return 0;
}
```

Again this entire program contains no novel code; so, we will not examine it line by line.

5 FileStream Management

5.1 Copy, Move, and Delete, but I How to Write!

As of this moment you may be thinking to yourself, "Well, this file management package is all fine and good. I can copy files, move files, delete file, and examine a file, but how do I write?" This question is answered by the file stream management package.

The file stream management package exists to allow the application programmer to read, write, and seek on a file on a byte-by-byte basis. This package allows the application programmer to read an arbitrary byte in a file, flip individual bits within a file, or seek to any point in a file with byte precision.

5.2 The FileStream Package

In C there exists a suite of functions which allows for the application programmer to read, write, and seek on a file on a byte-by-byte basis. Among the plethora of functions which exists in C to allow the application programmer this luxury are `fgets`, `fputs`, and `fseek`. This trinity of functions, however, limit the application programmer to reading, writing, and seeking only on the local hard drive, useful, but not as sexy as being able to read, write, and do seeks on files half-way across the world over secure protocols. This is the narcotic with which GAT provides the application programmer. The process of reading, writing, and seeking on a file through the aegis of GAT is accomplished through the use of a single class `GATFileStream`, the involution of which we shall next examine.

5.2.1 Constructing and Destroying FileStream Instances

The first step in doing anything at all with a `GATFileStream` is creating a `GATFileStream`. The is accomplished through a call to the function

```
GATFileStream GATFileStream_Create( GATContext context,
                                    GATPreferences_const preferences,
                                    GATLocation location,
                                    GATFileStreamMode mode)
```

The first argument is a `GATContext` instance the use of which was sketched in the previous chapter. The second argument is a `GATPreferences` instance which in most of our examples will be `NULL`, see Appendix D for the details of this class. The third argument is a `GATLocation` instance which specifies the location of the physical file which can be read, wrriten, and seeked upon by the resultant `GATFileStream`. The final argument is an enumeration `GATFileStreamMode`. The various values, and the associated semantics, for this enumeration are as follows

GATFileStreamMode Value	**Description of file state**
GATFileStreamMode_Read	Opened for reading from the file's beginning.
GATFileStreamMode_Write	Opened for writing from the file's beginning.
GATFileStreamMode_Append	Opened for writing from the file's ending.
GATFileStreamMode_ReadWrite	Opened for reading/writing from the file's beginning.

Table 2: GATFileStreamMode enumeration values

As one can see, the value of this enumeration governs the later use of the `GATFileStream`, if it is used for reading, writing, reading and writing, and where this reading or writing occurs. Finally, this function returns a `GATFileStream` instance corresponding to all the passed information.

As an example of the use of this creation function let us take a look at the process of creating a `GATFileStream` which could be used for reading or writing and is opened to read and write from the file's begining

```
GATContext context;
GATLocation location;
GATFileStream fileStream;

context = ...
location = ...

fileStream = GATFileStream_Create( context,
                                   NULL,
                                   location,
                                   GATFileStreamMode_ReadWrite );

if( NULL != fileStream )
{
  /* Use fileStream here */
}
```

Upon creating a `GATFileStream` instance and making it perform like a trained monkey for our pleasure we need to clean upon after the baboon-jester. This is accomplished through use of the function

```
void GATFileStream_Destroy(GATFileStream *strm)
```

This function takes as its first argument the `GATFileStream` which is to be destroyed. Upon return, this function releases all resources which the passed `GATFileStream` maintained.

5.2.2 Reading from a FileStream Instance

The class `GATFileStream` implements the interface `GATInterface_IStreamable`, as can bee seen in figure 17. It is through this interface that the application programmer can read, write, and seek on the physical file corresponding to the particular `GATFileStream` instance.

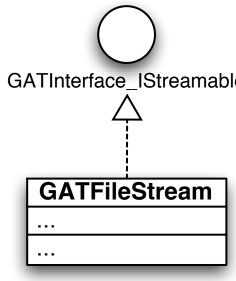

Figure 17: GATFileStream's realization of the interface GATInterface_IStreamable.

As mentioned in Chapter 3, there exist a set of functions corresponding to the GAT interface `GATInterface_IStreamable` which ease the task of the application programmer trying to make use this interface. As mentioned in the Chapter 3 this set of functions can loosely be thought of representing the interface `GATInterface_IStreamable`. This set of functions is given by the figure 18.

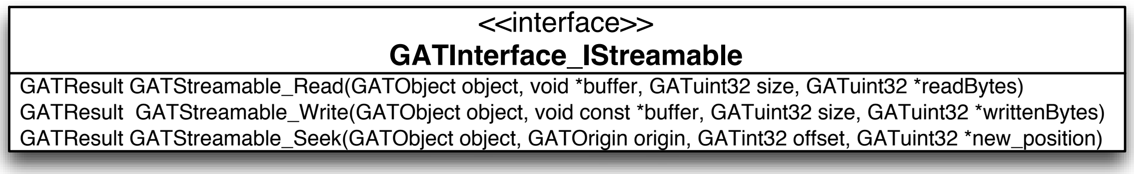

Figure 18: Utility functions for the interface GATInterface_IStreamable.

In reading from a `GATFileStream` instance we will make use of this set of utility functions instead of dealing directly with the interface `GATInterface_IStreamable` as, truthfully, its much easier.

So to read from a given `GATFileStream` instance we will make use of the function

```
GATResult GATStreamable_Read(GATObject object,
                             void *buffer,
                             GATuint32 size,
                             GATuint32 *read_bytes)
```

The first argument to this function is a `GATObject` instance. This `GATObject` instance must realize the interface `GATInterface_IStreamable`. For our immediate concerns this first object will always be an instance of a `GATFileStream`. The second argument to this function is a `void *` this pointer points to a buffer into which the read should occur. The next argument is a `GATuint32`, a primitive type covered in Appendix A, which passes the this function the length of the buffer in bytes. The final argument to this function is a `GATuint32 *` which passes the caller the actual number of bytes read. Finally, the function returns a `GATResult`, covered in Appendix C, which indicates the completion status of the function.

As an example, let us consider using the above function to read from a `GATFileStream` instance `fileStream` into a buffer `buffer` of size `bufferSize`. This call would look as follows

```
void *buffer;
GATResult result;
GATObject object;
GATuint32 readBytes;
GATuint32 bufferSize;
GATFileStream fileStream;

buffer = ...
bufferSize = ...

fileStream = ...

object = GATFileStream_ToGATObject( fileStream );

result = GATStreamable_Read( object, buffer, bufferSize, &readBytes );

if( GAT_SUCCEEDED( result ) )
{
  /* Do something with readBytes bytes in buffer */
}
```

5.2.3 Writing to a FileStream Instance

As the class `GATFileStream` implements the interface `GATInterface_IStreamable`, writing to a `GATFileStream` is as easy as reading from a `GATFileStream` instance. To do so we need only make use of the utility function

```
GATResult GATStreamable_Write(GATObject object,
                              void *buffer,
                              GATuint32 size,
                              GATuint32 *written_bytes)
```

The first argument to this function is a `GATObject` instance. This `GATObject` instance must realize the interface `GATInterface_IStreamable`. For our immediate goals this instance will always be an instance of a `GATFileStream`. The second argument to this function is a `void *` pointing to a buffer containing the data to be written. The next argument is a `GATuint32`, a GAT primitive type covered in Appendix A, which passes to the function the length of the buffer in bytes. The final argument is a `GATuint32 *` which passes back to the caller the actual number of bytes written. Finally, this function returns a `GATResult`, covered in Appendix C, which indicates the completion status of this function.

As a quick example let us write a buffer `buffer` filled with `sizeBuffer` bytes to a `GATFileStream` instance `fileStream`. Such a call would look as follows

```
void *buffer;
GATResult result;
```

```
GATObject object;
GATuint32 bufferSize;
GATuint32 writtenBytes;
GATFileStream fileStream;

buffer = ...
bufferSize = ...

fileStream = ...

object = GATFileStream_ToGATObject( fileStream );

result = GATStreamable_Write( object, buffer, bufferSize, &writtenBytes );

if( GAT_SUCCEEDED( result ) )
{
  /* Do something as writtenBytes of buffer have been written to fileStream */
}
```

5.2.4 Seeking on a FileStream Instance

As one may guess by now, as a result of the class `GATFileStream` realizes the interface `GATInterface_IStreamab` seeking on a `GATFileStream` is as easy as reading or writing to one. One need only make use of the proper utility function. In tis case it is

```
GATResult GATStreamable_Seek(GATObject object,
                             GATOrigin origin,
                             GATuint32 offset,
                             GATuint32 *new_position)
```

The first argument to this function is a `GATObject` instance. This `GATObject` instance must realize the interface `GATInterface_IStreamable`. For our immediate goals this instance will always be an instance of a `GATFileStream`. Do you have that strange sense déjà vu that I do? The second argument is an enumeration `GATOrigin` which indicates where this seek is to occur from. The possible values for the enumeration `GATOrigin` and their associated semantics are as follows

GATOrigin Value	**Description of seek semantcs**
GATOrigin_Set	Seek occurs from the beginning of the stream.
GATOrigin_Current	Seek occurs from the current stream position.
GATOrigin_End	Seek occurs from the end of the stream.

Table 3: GATOrigin enumeration values

The next argument to this function is a `GATuint32`, `offset`, indicating the number of bytes to position the current "cursor" from the specified `GATOrigin`. If `GATOrigin_Set` is the specified `GATOrigin`, then upon success the "cursor" is placed `offset` bytes after the stream's begining. If `GATOrigin_Current` is the specified `GATOrigin`, then upon success the "cursor" is placed `offset` bytes from the stream's current "cursor" position. If GATOrigin_End is the specified GATOrigin, then upon success the "cursor" is placed offset bytes before the stream's end. The final argument of this function is a GATuint32 * which passes back to the caller the new position of the "cursor" as a byte offset from the stream's beginning.

To get a feel for the use of this function, let us consider its use in seeking offset bytes from the beginning of a GATFileStream instance fileStream. The code which implements this little idea looks a bit like this

```
GATResult result;
GATuint32 offset;
GATObject object;
GATuint32 newPosition;
GATFileStream fileStream;

offset = ...

fileStream = ...

object = GATFileStream_ToGATObject( fileStream );

result = GATStreamable_Seek( object, GATOrigin_Set, offset, &newPosition );

if( GAT_SUCCEEDED( result ) )
{
  /* Do something as the cursor is at newPosition */
}
```

5.3 Some Useful Programs

Now that we have seen the majority of functionality provided by the GATFileStream class we are in a position to twist this class to our own deviant needs. We will do so, as in the last chapter, by creating a few example programs which make use of the various functions covered in this chapter. In particular, we will again try to create a few "grid enabled" version of now common command line tools.

5.3.1 From hexdump to gridhexdump in three easy steps!

The program hexdump is useful for those aberrant Unix hacks who find pleasure in dipping their ladle into the binary representation of various files. In its most common usage scenario this program takes a file, specified on the command line, and displays, in a human readable form, the binary data contained within the so specified file.

For example, if I have a file called /usr/home/example which contains only the text

```
Hello, cruel world!
```

then the command

```
% hexdump /usr/home/example
```

will print out

```
0000000 4865 6c6c 6f2c 2063 7275 656c 2077 6f72
0000010 6c64 210a
0000014
```

Looks like a mess you say; well it is human readable, you just need to know the secret handshake.

The default format that `hexdump` displays this binary data in is called the "two-byte hexadecimal display" and is what you see above. From the manual for `hexdump` it "displas the input offset in hexadecimal, followed by eight, space separated, four column, zero-filled, two-byte quantities of input data, in hexadecimal, per line." So the first number `0000000` of the line

```
0000000 4865 6c6c 6f2c 2063 7275 656c 2077 6f72
```

indicates that the byte offset of the hexadecimal number `4865` from the beginning of the file is `0000000` bytes. Similarly, the line

```
0000010 6c64 210a
```

indicates that the byte offset of the hexadecimal number `6c64` from the beginning of the file is `0000010` bytes, remember this number is in hexadecimal, its `16` in binary.

So, for example, in looking at the original text in `/usr/home/example`

```
Hello, cruel world!
```

along with the output of our `hexdump` call

```
0000000 4865 6c6c 6f2c 2063 7275 656c 2077 6f72
0000010 6c64 210a
0000014
```

we can see that the hexadecimal character code for 'H' is 48, the hexadecimal character code for 'e' is 65, the hexadecimal character code for 'l' is 6*c*, . . . The final number

```
0000014
```

is the total number of bytes in the file. See, after you learn the secret handshake it's not really all that hard.

Now lets move on to make a "grid enabled" version of `hexdump`. The full code for a program which does the trick is as follows

```
#include <stdio.h>
#include "GAT.h"

int main( int argc, char *argv[] )
{
  int counter;
  char buffer[16];
```

```
GATuint32 offset;
GATContext context;
GATuint32 readBytes;
GATLocation location;
GATFileStream fileStream;
GATObject fileStreamObject;

/* Check command line syntax */
if( 2 != argc )
{
   printf("usage: %s file\n", argv[0]);

   return 1;
}

/* Set result to a memory failure */
result = GAT_MEMORYFAILURE;

/* Create a GATLocation location */
location = GATLocation_Create( argv[1] );

/* Check GATLocation creation */
if( NULL != location )
{
  /* Create GATContext context */
  context = GATContext_Create();

  /* Check GATContext creation */
  if( NULL != context )
  {
    /* Create GATFileStream fileStream */
    fileStream  = GATFileStream_Create( context,
                                        NULL,
                                        location,
                                        GATFileStreamMode_Read );

    /* Check GATFileStream creation */
    if( NULL != fileStream )
    {
      /* Set offset */
      offset = 0;

      /* Cast GATFileStream to GATObject */
      fileStreamObject = GATFileStream_ToGATObject( fileStream );

      /* Read in 16 bytes */
      result =GATStreamable( fileStreamObject, (void *) buffer, 16, &readBytes );

      /* Loop until a read failure */
```

```
      while( GAT_SUCCEEDED( result ) )
      {
        /* Print out offset */
        printf( "%07x ", offset );

        /* Print out data */
        for( count = 0; count < readBytes; count++ )
        {
          printf( "%02x", buffer[count] );

          if( 0 == ( (count + 1) % 2 ) )
          {
            printf(" ");
          }
        }

        /* Print newline character */
        printf( "\n" );

        /* Set offset */
        offset = offset + readBytes;

        /* Read in 16 bytes */
        result =GATStreamable( fileStreamObject, (void *) buffer, 16, &readBytes );
      }

      /* Destroy GATFileStream */
      GATFileStream_Destroy( &fileStream );
    }

    /* Destroy GATContext */
    GATContext_Destroy( &context );
  }

  /* Print the total number of bytes */
  if( 0 != offset )
  {
    printf( "%07x\n", offset );
  }

  /* Destroy GATLocation */
  GATLocation_Destroy( &location );
}

/* Check result for success and print error */
if( GAT_FAILED( result) )
{
  printf( "An error has occurred\n");
```

```
    return 1;
  }

  return 0;
}
```

As we have reviewed all the various functions in this program previously, we will not belabor the point by reviewing them again here.

5.3.2 A cp from the ground up

Next we will revisit, with a slight variation, the theme of `cp`. Previously we had written a "grid enabled" `cp` which used the class `GATFile` to copy a file from point A to point B. In this next example we will rewrite this program to using `GATFileStream` instead of the class `GATFile` to get the job done. This process if involve copying, byte by byte, the entire source file to the destination. Here's the code

```
#include <stdio.h>
#include "GAT.h"

int main( int argc, char *argv[] )
{
  char buffer[32];
  GATResult result;
  GATuint32 readBytes;
  GATContext context;
  GATuint32 writtenBytes;
  GATLocation sourceLocation;
  GATFileStream sourceFileStream;
  GATLocation destinationLocation;
  GATFileStream destinationFileStream;
  GATObject sourceFileStreamObject;
  GATObject destinationFileStreamObject;


  /* Check command line syntax */
  if( 3 != argc )
  {
     printf("usage: %s source destination\n", argv[0]);

     return 1;
  }

  /* Set result to a memory failure */
  result = GAT_MEMORYFAILURE;

  /* Create GATLocation sourceLocation */
  sourceLocation = GATLocation_Create( argv[1] );
```

```
/* Check previous GATLocation creation */
if( NULL != sourceLocation )
{
  /* Create GATLocation destinationLocation */
  destinationLocation = GATLocation_Create( argv[2] );

  /* Check previous GATLocation creation */
  if( NULL != destinationLocation )
  {
    /* Create GATContext context */
    context = GATContext_Create();

    /* Check previous GATContext creation */
    if( NULL != context )
    {
      /* Create GATFileStream sourceFileStream */
      sourceFileStream = GATFileStream_Create( context,
                                               NULL,
                                               sourceLocation,
                                               GATFileStreamMode_Read );

      /* Check GATFileStream creation */
      if( NULL != sourceFileStream )
      {
        /* Create GATFileStream destinationFileStream */
        destinationFileStream = GATFileStream_Create( context,
                                                      NULL,
                                                      destinationLocation,
                                                      GATFileStreamMode_Write );

        /* Check GATFileStream creation */
        if( NULL != destinationFileStream )
        {
          /* Convert  sourceFileStream to a GATObject */
          sourceFileStreamObject =
            GATFileStream_ToGATObject( sourceFileStream );

          /* Convert  destinationFileStream to a GATObject */
          destinationFileStreamObject =
            GATFileStream_ToGATObject( destinationFileStream );

          /* Read from sourceFileStreamObject */
          result =
            GATStreamable_Read( sourceFileStreamObject,
                                (void *) buffer,
                                32,
                                &readBytes );

          /* Loop until done */
```

```
          while( GAT_SUCCEEDED( result ) )
          {
            /* Write to destinationFileStreamObject */
            result =
              GATStreamable_Write( destinationFileStreamObject,
                                   (void *) buffer,
                                   readBytes,
                                   &writtenBytes );

            /* Read from sourceFileStreamObject */
            if( GAT_SUCCEEDED( result ) )
            {
              result =
                GATStreamable_Read( sourceFileStreamObject,
                                    (void *) buffer,
                                    32,
                                    &readBytes );
            }
          }

          /* Destroy GATFileStream destinationFileStream */
          GATFileStream_Destroy( &destinationFileStream );
        }

        /* Destroy GATFileStream sourceFile */
        GATFileStream_Destroy( &sourceFileStream );
      }

      /* Destroy GATContext context */
      GATContext_Destroy( &context );
    }

    /* Destroy GATLocation destinationLocation */
    GATLocation_Destroy( &destinationLocation );
  }

  /* Destroy GATLocation sourceLocation */
  GATLocation_Destroy( &sourceLocation );
}

/* Check result for success and print error */
if( GAT_FAILED( result) )
{
  printf( "An error has occurred during the copy operation\n");

  return 1;
}

return 0;
```

```
}
```

We will not belabor the the code above by reviewing it line by line, take it as a homework assignment.

5.3.3 gridhexdump, theme and variation

Earlier we gave a code example which created a sort of "gridhexdump," implementing the basic functionality of the command line program `hexdump`. However the actual program `hexdump` is a bit more complicated than the one we gave. In particular, the program `hexdump` allows for various command line options which modify the basic functionality of `hexdump`. In this example will expand our `gridhexdump` to take one of these command line options.

The program `hexdump` has a command line option `-s` which allows for the user to stipulate where in the specified file `hexdump` should start reading from. For example the command

```
% hexdump -s 6 /usr/home/example
```

would case the program `hexdump` to start reading data from the 6th byte in the file `/usr/home/example`. While the command

```
% hexdump -s 4 /usr/home/example
```

would case the program `hexdump` to start reading data from the 4th byte in the file `/usr/home/example`.

If our example file is the same as before

```
Hello, cruel world!
```

then the command

```
% hexdump -s 4 /usr/home/example
```

would yield

```
0000004 6f2c 2063 7275 656c 2077 6f72 6c64 210a
0000014
```

We will add this optional command to our `hexdump` in the next code example. The full code for this little trick is below

```
#include <stdio.h>
#include <stdlib.h>
#include "GAT.h"

int main( int argc, char *argv[] )
{
  int counter;
  char buffer[16];
  GATuint32 offset;
```

```
GATContext context;
GATuint32 readBytes;
GATLocation location;
GATuint32 newPosition;
GATFileStream fileStream;
GATObject fileStreamObject;

/* Check command line syntax */
if( (4 != argc) && (2 != argc) )
{
   printf("usage: %s [-s N] file\n", argv[0]);

   return 1;
}

/* Set result to a memory failure */
result = GAT_MEMORYFAILURE;

/* Create a GATLocation location */
if( 2 == argc )
{
  location = GATLocation_Create( argv[1] );
}
else
{
  location = GATLocation_Create( argv[3] );
}

/* Check GATLocation creation */
if( NULL != location )
{
  /* Create GATContext context */
  context = GATContext_Create();

  /* Check GATContext creation */
  if( NULL != context )
  {
    /* Create GATFileStream fileStream */
    fileStream  = GATFileStream_Create( context,
                                        NULL,
                                        location,
                                        GATFileStreamMode_Read );

    /* Check GATFileStream creation */
    if( NULL != fileStream )
    {
      /* Set offset */
      if( 2 == argc )
      {
```

```
  offset = 0;
}
else
{
  offset = (GATuint32) atol( argv[2] );
}

/* Cast GATFileStream to GATObject */
fileStreamObject = GATFileStream_ToGATObject( fileStream );

/* Seek to offset */
result = GATStreamable_Seek( fileStreamObject, GATOrigin_Set, offset, &newPosition )

/* Check previous seek */
if( GAT_SUCCEEDED( result ) )
{

  /* Read in 16 bytes */
  result = GATStreamable_Read( fileStreamObject, (void *) buffer, 16, &readBytes );

  /* Loop until a read failure */
  while( GAT_SUCCEEDED( result ) )
  {
    /* Print out offset */
    printf( "%07x ", offset );

    /* Print out data */
    for( count = 0; count < readBytes; count++ )
    {
      printf( "%02x", buffer[count] );

      if( 0 == ( (count + 1) % 2 ) )
      {
        printf(" ");
      }
    }

    /* Print newline character */
    printf( "\n" );

    /* Set offset */
    offset = offset + readBytes;

    /* Read in 16 bytes */
    result =GATStreamable( fileStreamObject, (void *) buffer, 16, &readBytes );
  }

}
```

```
        /* Destroy GATFileStream */
        GATFileStream_Destroy( &fileStream );
      }

      /* Destroy GATContext */
      GATContext_Destroy( &context );
    }

    /* Print the total number of bytes */
    if( 0 != offset )
    {
      printf( "%07x\n", offset );
    }

    /* Destroy GATLocation */
    GATLocation_Destroy( &location );
  }

  /* Check result for success and print error */
  if( GAT_FAILED( result) )
  {
    printf( "An error has occurred\n");

    return 1;
  }

  return 0;
}
```

Again as we have reviewed all the various functions in this program previously, we will not belabor the point by reviewing them again here.

6 LogicalFile Management

6.1 The Shortest Distance Between Two Points...

Grid computing introduces a hateful mess of complications which lie beyond the pail of more conventional computing models. Out of this thickety mess the LogicalFile package clears the brush of a single problem. Within the realm of grid computing, it often is the case that a user has identical files distributed throughout the geography of network space. In this very situation, almost in an attempt to make things worse, the user often has the desire to make even more copies of this Dolly the file; the doppelgangers multiply. This, at first, may not seem such a problem. However, the rub is that these files are often immense and would choke the puny networks linking these computers, taking forever to copy from here to there. The LogicalFile package tries to ameliorate this very problem.

The LogicalFile package does this by abstracting the process of copying the file away from the application programmer and user, then putting a little smarts in to the operation. GAT and it minions decide which of the various files is closest in this geography of network space to the target location, then uses this file to make the new copy so that this process of asexual reproduction proceeds as efficiently as is possible.

6.2 The LogicalFile Package

The LogicalFile package consists, quite economically, of a single class `GATLogicaFile` which does all the heavy lifting. A `GATLogicaFile` instance represents a set of physical files which are byte-for-byte identical but dispersed throughout the geography of network space. This `GATLogicaFile` is useful for the very case described above. A `GATLogicaFile` instance, representing a set of identical physical files, abstracts the process of deciding which of the physical files away from the user and application programmer thus lifting the burden of determining which of these many files is closest in this network geography to the desired target location. Let begin our study of this beast.

6.2.1 Constructing and Destroying LogicalFile Instances

As always, before we can run we must lean how to walk. Our first step is thus learning how to create a `GATLogicalFile` instance. This is done through a call to the function

```
GATLogicalFile
  GATLogicalFile_Create(GATContext context, GATLocation location,
```

```
    GATLogicalFileMode mode, GATPreferences_const preferences)
```

The first argument to this function is a `GATContext`, covered previously in this manual. The next argument is a `GATLocation` instance. This `GATLocation` instance is simply used as a name for the resultant `GATLogicalFile` instance, it has nothing do do with actual physical location. The next argument is the enumeration `GATLogicalFileMode`. The various values of this enumeration dictate what should occur if the specified `GATLocation` corresponds to a preexisting `GATLogicalFile` instance. The various values that this enumeration may take on are given, along with their semantics, in the following table

GATLogicalFileMode Value	**Resultant create semantics**
GATLogicalFileMode_Open	Open the GATLogicalFile if it exists
GATLogicalFileMode_Create	Create the GATLogicalFile if it does not exist
GATLogicalFileMode_Truncate	Create the GATLogicalFile if it does or does not exist

Table 4: GATLogicalFileMode enumeration values

The final argument to this function is a `GATPreferences` instance, covered in Appendix D. This function returns a `GATLogicalFile` corresponding to the passed information and NULL upon failure.

As an example, let us consider opening a `GATLogicalFile` named by the `GATLocation` instance `location`. A code snippet which fits the bill is

```
GATContext context;
GATLocation location;
GATLogicalFile logicalFile;

context = ...
location = ...

logicalFile = GATLogicalFile_Create( context, location, GATLogicalFileMode_Open, NULL );

if( NULL != logicalFile )
{
  /* Play GATLogicalFile games */
}
```

After creating a `GATLogicalFile` and playing any amusing games with it that may cross our minds we need to send it out to pasture. This is accomplished through a call to the "executioner"

```
void GATLogicalFile_Destroy(GATLogicalFile *logicalFile)
```

This function takes as tis first and only argument the `GATLogicalFile` which is to be destroyed. Upon return from this function all resources which the passed `GATLogicalFile` maintained are freed.

6.2.2 Adding and Removing File Instances

Upon creating a `GATLogicalFile` instance, for it to be of any use it must contain reference to byte-for-byte identical physical files. So, there has to be some means of adding, and removing, references to physical files. In this section we will take a look at how to add and remove reference to physical files.

In GAT physical files are represented through `GATFile` instances. Hence to add a reference to a physical file to a `GATLogicalFile` instance we need to associate this `GATLogicalFile` instance with a `GATFile` instance. This is done through a call to the following function

```
GATResult GATLogicalFile_AddFile(GATLogicalFile logicalFile, GATFile_const file)
```

The first argument to this function is a `GATLogicalFile`, the `GATLogicalFile` which is to be added to. The next argument is a `GATFile_const`. This instance represents the physical file which is be associated with the `GATLogicalFile` instance upon successful completion of this function. The return value of this function is a `GATResult`, covered in Appendix C, which indicates the completion status of this function.

To animate this still-life lets look at this function is use. Consider associating a `GATLogicalFile` instance `logicalFile` with a `GATFile` instance `file`. A code with performs this little pirouette on command is as follows

```
GATFile file;
GATResult result;
GATLogicalFile logicalFile;

file = ...
logicalFile = ...

result = GATLogicalFile_AddFile( logicalFile, file );

if( GAT_SUCCEEDED( result ) )
{
  /* The file has been added to the logicalFile */
}
```

Often it is the case that, well, we simply change our mind or need to modify one of the physical files contained in a a logical file so that it's not byte-for-byte identical to the others. In these cases, and others, we need to remove a physical file from a given `GATLogicalFile`. This is accomplished through a call to the function

```
GATResult
  GATLogicalFile_RemoveFile(GATLogicalFile logfile, GATFile_const file)
```

The first argument to this function is a `GATLogicalFile` instance containing the `GATLogicalFile` which is to be modified. The next argument is a `GATFile_const` instance identifying the `GATFile` instance which is to be removed, this passed `GATFile` instance will match and thus remove a contained `GATFile` instance is and only if the two instances return a `GATrue` when passed to the "Equals" function of `GATFile`. Finally, this function returns a `GATResult`, described in Appendix

C, which indicates it completion status.

To get a better feel for this function in use, consider the code for removing from a `GATLogicalFile` instance `logicalFile` a `GATFile` instance `file`. The code with performs this little task takes the form

```
GATFile file;
GATResult result;
GATLogicalFile logicalFile;

file = ...
logicalFile = ...

result = GATLogicalFile_RemoveFile( logicalFile, file );

if( GAT_SUCCEEDED( result ) )
{
  /* The file has been removed from the logicalFile */
}
```

6.2.3 Replicating LogicalFile Instances

The raison d'être for the existence of the `GATLogicalFile`, like that for most biological organisms, is to go forth and multiply. This process of asexual replication occurs for the `GATLogicalFile` species is accomplished through a call to the function

```
GATResult
  GATLogicalFile_Replicate( GATLogicalFile_const logfile,
    GATLocation_const target)
```

The first argument to this function is a `GATLogicalFile_const` instance which is the `GATLogicalFile_const` containing the physical files that one wishes to replicate. The next argument to this function is a `GATLocation_const` instance which qualifies the location of the replica. Finally, this function returns a `GATResult`, covered in Appendix C, which indicates the return status of this function.

To take this function out for a test drive we can should how it would be used to replicate a `GATLogicalFile` instance `logicalFile` to a `GATLocation` instance `location`. In code this spin around the block looks as follows

```
GATResult result;
GATLocation location;
GATLogicalFile logicalFile;

location = ...
logicalFile = ...

result = GATLogicalFile_Replicate( logicalFile, location );

if( GAT_SUCCEEDED( result ) )
{
```

```
  /* The asexual reproduction has succeeded! */
}
```

6.2.4 Examining LogicalFile Instances

In addition to the above tricks, adding files, removing files, and replicating files, the application programmer can also inspect `GATLogicalFile` instances. In particular, an application programmer can obtain, from a `GATLogicalFile` instance, a list of the various `GATFile` this `GATLogicalFile` "contains". This is accomplished through a call to the function

```
GATResult
  GATLogicalFile_GetFiles(GATLogicalFile_const logfile,
    GATList_GATFile *files)
```

The first argument to this function is a `GATLogicalFile` instance. This instance is the one to be examined. The next argument is a `GATList_GATFile`. !!!. This passes back to the caller a list containing `GATFile` instances represented by the specified `GATLogicalFile` instance. This function returns a `GATResult`, covered in Appendix C, which indicates the completion status of this function.

For the test drive this time we'll obtain a `GATList_GATFile` containing the `GATFile` instances in a `GATLogicalFile` instance `logicalFile`. The code for this is

```
GATResult result;
GATList_GATFile fileList;
GATLogicalFile logicalFile;

logicalFile = ...

result = GATLogicalFile_GetFiles( logicalFile, &fileList );

if( GAT_SUCCEEDED( result ) )
{
  /* The fileList is now populated with GATFile's */
}
```

6.3 Some Useful Programs

6.3.1 Put 'dat in da girdbag!

As the problem which the class `GATLogicalFile` solves does not arise commonly in the Unix world, there are no common command line tools which we can ape that would flex the muscle of the class `GATLogicalFile`. So, we'll have to invent our own. Let us christen her `gridbag`.

The command line tool `gridbag` is used to create `GATLogicalFile` instances, examine `GATLogicalFile` instances, and to add/remove files from a `GATLogicalFile` instance. To create a `GATLogicalFile` instance with a given name the `gridbag` tool command line syntax will be as follows

```
% gridbag logicalfile
```

This command will create, if it does not already exist, a `GATLogicalFile` with the name `logicalfile`. After a `GATLogicalFile` with a given name has been created, our `gridbag` tool can also be used to examine its contents. For example, to list the contents of the pre-existing `GATLogicalFile` instance with the name `logicalfile` the `gridbag` tool can be called as follows

```
% gridbag logicalfile
```

Such a call will print out a list of the various `GATFile` instances contained within the named `GATLogicalFile` instance. This would, for example, look something like this

```
http://www.google.com/index.html
http://arts.ucsc.edu/faculty/cope/index.htm
http://hussle.harvard.edu/~pyesley/chickens.html
http://www.shutemdown.com/pebtn2000.htm
```

In addition our, as of yet fictional `gridbag` tool can also be used to add files to an existing `GATLogicalFile` instance. If there exists a `GATLogicalFile` instance named `logicalfile`, then we could add the physical file named `file` to this `GATLogicalFile` instance through the following call to the mighty `gridbag`

```
% gridbag logicalfile -a file
```

Similarly we can remove files from a pre-existing `GATLogicalFile` instance through use of `gridbag`. If there exists a `GATLogicalFile` instance named `logicalfile`, then we could remove the physical file named `file` from this `GATLogicalFile` instance through the following call to the `gridbag`

```
% gridbag logicalfile -r file
```

Now. let us produce the source code for this as of yet wholly fictitious command line tool. It is given as follows

```
#include <stdio.h>
#include <string.h>
#include "GAT.h"

static int addToLogicalFile( argc, argv );
static int examineLogicalFile( argc, argv );
static int removeFromLogicalFile( argc, argv );
static int examineOrCreateLogicalFile( argc, argv );
static int commandLineArgumentsValid( int argc, char *argv[] );

int main( int argc, char *argv[] )
{
  int result;

  /* Check command line arguments */
  if( 0 != (result = commandLineArgumentsValid(argc, argv)) )
  {
    /* Print out usage information */
```

```
    printf( "usage: gridbag logicalfile [ -a file | -r file ]\n" );

    /* Return to OS */
    return result;
  }

  /* Handle examination and creation cases */
  if( 2 == argc )
  {
    if( 0 != (result = examineOrCreateLogicalFile(argc, argv)) )
    {
      printf( "An error has occured.\n" );
    }

    /* Return to OS */
    return result;
  }

  /* Handle add case */
  if( 0 == strcmp(argv[2], "-a") )
  {
    if( 0 != (result = addToLogicalFile(argc, argv)) )
    {
      printf( "An error has occured.\n" );
    }

    /* Return to OS */
    return result;
  }

  /* Handle remove case */
  if( 0 != (result = removeFromLogicalFile(argc, argv)) )
  {
    printf( "An error has occured.\n" );
  }

  /* Return to OS */
  return result;
}

static int commandLineArgumentsValid( int argc, char *argv[] )
{
  int commandLineArgumentsValid;

  /* Assume invalid arguments */
  commandLineArgumentsValid = 0;

  /* Check examination and creation cases */
  if( 2 == argc )
```

```
  {
    commandLineArgumentsValid = 1;
  }

  /* Check addition and removal cases */
  if( 4 == argc )
  {
    /* Check addition case */
    if( 0 == strcmp(argv[2], "-a") )
    {
      commandLineArgumentsValid = 1;
    }

    /* Check removal case */
    if( 0 == strcmp(argv[2], "-r") )
    {
      commandLineArgumentsValid = 1;
    }
  }

  /* Return to caller */
  return commandLineArgumentsValid;
}

static int examineOrCreateLogicalFile( argc, argv )
{
  int result;
  GATContext context;
  GATLocation location;
  GATLogicalFile logicalFile;


  /* Create GATContext */
  context = GATContext_Create();

  /* Check GATContext creation */
  if( NULL != context )
  {
    /* Create GATLocation */
    location = GATLocation_Create( argv[1] );

    /* Check GATLocation creation */
    if( NULL != location )
    {
      /* Create brand new GATLogicalFile  */
      logicalFile =
        GATLogicalFile_Create( context,
                               location,
                               GATLogicalFileMode_Create,
```

```
                              NULL );

      /* Destroy GATLocation */
      GATLocation_Destroy( &location );
    }

    /* Destroy GATContext */
    GATContext_Destroy( &context );
  }

  /* Check GATLogicalFile creation */
  if( NULL !=  logicalFile )
  {
    /* Creation succeeded */
    result = 0;
  }
  else
  {
    /* Creation failed, assume named GATLogicalFile exists and examine it */
    result = examineLogicalFile( argc, argv );
  }

  /* Return to caller */
  return result;
}

static int examineLogicalFile( argc, argv )
{
  int result;
  GATFile currentFile;
  GATResult gatResult;
  GATContext context;
  GATLocation location;
  GATList_GATFile filesList;
  GATLogicalFile logicalFile;
  GATLocation_const currentLocation;
  GATList_GATFile_Iterator endIterator;
  GATList_GATFile_Iterator currentIterator;


  /* Set result, assume failure */
  result = 1;

  /* Create GATContext */
  context = GATContext_Create();

  /* Check GATContext creation */
  if( NULL != context )
  {
```

```
/* Create GATLocation */
location = GATLocation_Create( argv[1] );

/* Check GATLocation creation */
if( NULL != location )
{
  /* Open existing GATLogicalFile  */
  logicalFile = GATLogicalFile_Create( context,
                                       location,
                                       GATLogicalFileMode_Open,
                                       NULL );

  /* Check GATLogicalFile opening */
  if( NULL != logicalFile )
  {
    /* Obtain GATFiles */
    gatResult = GATLogicalFile_GetFiles( logicalFile, &filesList );

    /* Check GATLogicalFile_GetFiles call */
    if( GAT_SUCCEEDED( gatResult ) )
    {
      /* Obtain GATList_GATFile_Iterator for End */
      endIterator = GATList_GATFile_End( filesList );

      /* Check GATList_GATFile_Iterator */
      if( NULL != endIterator )
      {
        /* Obtain GATList_GATFile_Iterator for Begin */
        currentIterator = GATList_GATFile_End( filesList );

        /* Check GATList_GATFile_Iterator */
        if( NULL != currentIterator )
        {
          /* Loop over GATFile instances */
          while( endIterator != currentIterator )
          {
            /* Obtain current GATFile */
            currentFile = GATList_GATFile_Get( currentIterator );

            /* Check current GATFile */
            if( NULL != currentFile )
            {
              /* Obtain current GATLocation */
              currentLocation = GATFile_GetLocation( currentFile );

              /* Print out the currentLocation */
              printf( "%s\n", GATLocation_ToString( currentLocation ) );

              /* Increment currentIterator */
```

```
                currentIterator = GATList_GATFile_Next( currentIterator );
              }
            }

            /* Set result, assume success */
            result = 0;
          }
        }

        /* Destroy GATList_GATFile */
        GATList_GATFile_Destroy( &filesList );
      }

      /* Destroy GATLogicalFile */
      GATLogicalFile_Destroy( &logicalFile );
    }

    /* Destroy GATLocation */
    GATLocation_Destroy( &location );
  }

  /* Destroy GATContext */
  GATContext_Destroy( &context );
}

/* Return to caller */
return result;
}

static int addToLogicalFile( argc, argv )
{
  int result;
  GATFile file;
  GATResult gatResult;
  GATContext context;
  GATLogicalFile logicalFile;
  GATLocation fileLocation;
  GATLocation logicalFileLocation;

  /* Set gatResult */
  gatResult = GAT_MEMORYFAILURE;

  /* Create GATContext */
  context = GATContext_Create();

  /* Check GATContext creation */
  if( NULL != context )
  {
    /* Create GATLocation */
```

```
  logicalFileLocation = GATLocation_Create( argv[1] );

  /* Check GATLocation creation */
  if( NULL != logicalFileLocation )
  {
    /* Open GATLogicalFile  */
    logicalFile =
      GATLogicalFile_Create( context,
                             logicalFileLocation,
                             GATLogicalFileMode_Open,
                             NULL );

     /* Check GATLogicalFile creation */
     if( NULL != logicalFile )
     {
       /* Create GATLocation */
       fileLocation = GATLocation_Create( argv[3] );

       /* Check GATLocation creation */
       if( NULL != fileLocation )
       {
         /* Create GATFile */
         file = GATFile_Create( context, fileLocation, NULL );

         /* Check GATFile creation */
         if( NULL != file )
         {
           /* Add GATFile to GATLogicalFile */
           gatResult =  GATLogicalFile_AddFile( logicalFile, file );

           /* Destroy GATFile */
           GATFile_Destroy( &file );
         }

         /* Destroy GATLocation */
         GATLocation_Destroy( &fileLocation );
       }

       /* Destroy GATLogicalFile */
       GATLogicalFile_Destroy( &logicalFile );
     }

    /* Destroy GATLocation */
    GATLocation_Destroy( & logicalFileLocation );
  }

  /* Destroy GATContext */
  GATContext_Destroy( &context );
}
```

```
  /* Set result */
  result = 1;
  if( GAT_SUCCEEDED( gatResult ) )
  {
    result = 0;
  }

  /* Return to caller */
  return result;
}

static int removeFromLogicalFile( argc, argv )
{
  int result;
  GATFile file;
  GATResult gatResult;
  GATContext context;
  GATLogicalFile logicalFile;
  GATLocation fileLocation;
  GATLocation logicalFileLocation;

  /* Set gatResult */
  gatResult = GAT_MEMORYFAILURE;

  /* Create GATContext */
  context = GATContext_Create();

  /* Check GATContext creation */
  if( NULL != context )
  {
    /* Create GATLocation */
    logicalFileLocation = GATLocation_Create( argv[1] );

    /* Check GATLocation creation */
    if( NULL != logicalFileLocation )
    {
      /* Open GATLogicalFile  */
      logicalFile =
        GATLogicalFile_Create( context,
                               logicalFileLocation,
                               GATLogicalFileMode_Open,
                               NULL );

       /* Check GATLogicalFile creation */
       if( NULL != logicalFile )
       {
         /* Create GATLocation */
         fileLocation = GATLocation_Create( argv[3] );
```

```
        /* Check GATLocation creation */
        if( NULL != fileLocation )
        {
          /* Create GATFile */
          file = GATFile_Create( context, fileLocation, NULL );

          /* Check GATFile creation */
          if( NULL != file )
          {
            /* Remove GATFile from GATLogicalFile */
            gatResult =  GATLogicalFile_RemoveFile( logicalFile, file );

            /* Destroy GATFile */
            GATFile_Destroy( &file );
          }

          /* Destroy GATLocation */
          GATLocation_Destroy( &fileLocation );
        }

        /* Destroy GATLogicalFile */
        GATLogicalFile_Destroy( &logicalFile );
      }

      /* Destroy GATLocation */
      GATLocation_Destroy( & logicalFileLocation );
    }

    /* Destroy GATContext */
    GATContext_Destroy( &context );
  }

  /* Set result */
  result = 1;
  if( GAT_SUCCEEDED( gatResult ) )
  {
    result = 0;
  }

  /* Return to caller */
  return result;
}
```

As we have covered all the various functions using in `girdbag` individually we will not review each line of the above program here.

6.3.2 Form Dolly to griddolly

Our above Frankenstein, `gridbag`, actually is a useful grid tool. However, for all its utility it does not allow one to replicate a `GATLogicalFile` instance. With out next command line tool/example will will remedy this want with the introduction of the mighty tool `griddolly`.

The command line tool to-be `griddolly` allows one to take a pre-existing `GATLogicalFile` instance and replicate it to any given location. The command line syntax for this tool is as follows

```
% griddolly logicalfile location
```

The first command line argument `logicalfile` specifies the `GATLogicalFile` which is to be replicated. The second command line argument `location` specifies to where the `GATLogicalFile` instance named `logicalfile` is to be replicated. The full code for this example is as follows

```
#include <stdio.h>
#include "GAT.h"

int main( int argc, char *argv[] )
{
  int osResult;
  GATResult result;
  GATContext context;
  GATLogicalFile logicalFile;
  GATLocation replicaLocation;
  GATLocation logicalFileLocation;

  /* Check command line syntax */
  if( 3 != argc )
  {
    printf( "usage: griddolly logicalfile location\n" );

    return 1;
  }

  /* Set result */
  result = GAT_MEMORYFAILURE;

  /* Create GATContext */
  context = GATContext_Create();

  /* Check GATContext creation */
  if( NULL != context)
  {
    /* Create GATLocation */
    logicalFileLocation = GATLocation_Create( argv[1] );

    /* Check GATLocation creation */
    if( NULL != logicalFileLocation )
```

```
    {
      /* Create GATLogicalFile */
      logicalFile = GATLogicalFile_Create( context,
                                           logicalFileLocation,
                                           GATLogicalFileMode_Open,
                                           NULL );

      /* Check GATLogicalFile creation */
      if( NULL != logicalFile )
      {
        /* Create GATLocation */
        replicaLocation = GATLocation_Create( argv[2] );

        /* Check GATLocation creation */
        if( NULL != replicaLocation )
        {
          /* Replicate GATLogicalFile */
          result = GATLogicalFile_Replicate( logicalFile, replicaLocation );

          /* Destroy GATLocation */
          GATLocation_Destroy( &replicaLocation );
        }

        /* Destroy GATLogicalFile */
        GATLogicalFile_Destroy( &logicalFile );
      }

      /* Destroy GATLocation */
      GATLocation_Destroy( &logicalFileLocation );
    }

    /* Destroy GATContext */
    GATContext_Destroy( &context );
  }

  /* Set osResult  and print error message */
  osResult = 0;
  if( GAT_FAILED( result ) )
  {
    osResult = 1;

    printf( "There was an error in replication.\n" );
  }

  /* Return to OS */
  return osResult;
}
```

7 Advert Management

7.1 So, where did I put my Keys?

Wouldn't it be nice if each time you lost your keys you could summon your own personal djinni and bend her will to the task of key reconnaissance. Alas, at least as far as your humble author knows, there is no such djinni, beyond the TV djinni so fatefully portrayed in "I Dream of Jinni."

In the world of bits GAT, however, summons just such a djinni to your service. This djinni, instead of inhabiting a magic lamp dug up from the sandy shores of some distant Arabia, is held within the advertisement package.

7.2 The Advertisement Package

This digital djinni housed within the advertisement package consists of two constituents. The first is the interface `GATInterface_IAdvertisable` and the second, the class `GATAdvertService`. The interface is the keys and the class the djinni. Let me explain. . .

The interface `GATInterface_IAdvertisable` is an interface which is implemented by many classes within the GAT universe. In fact the full spectrum of such classes is shown in figure 19.

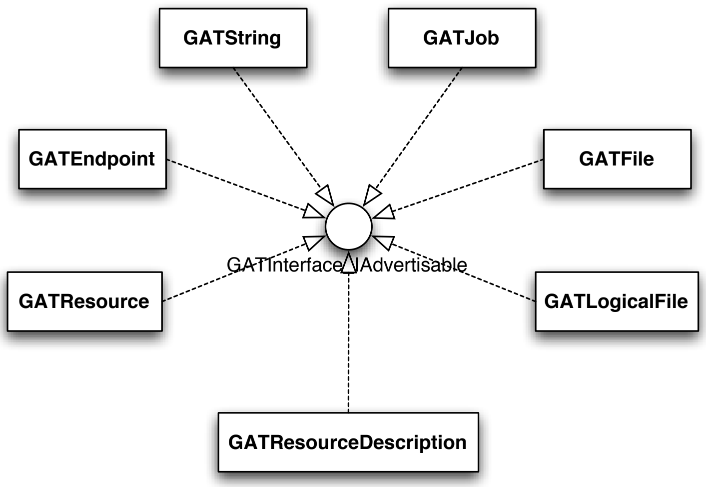

Figure 19: Classes realizing the interface GATInterface_IAdvertisable.

An instance of any class which implements this interface `GATInterface_IAdvertisable` can be placed in an `GATAdvertService` instance along with a description describing the advertisable. After this donation to the `GATAdvertService` djinni, whenever a user wants to find this deposited instance all they need do is give the djinni `GATAdvertService` a description of the instance they desire and this djnni will return a list of all the advertisables which fit this description. Magic, no? Furthermore, this `GATAdvertService` persists these advertisables and is accessible across machine boundaries. So, code on machine A can place an advertisable in a `GATAdvertService` on Tuesday, then next month code on machine B can remove or examine this same advertisable. Now that's really magic.

7.2.1 Constructing and Destroying AdvertService Instances

Before we can proffer up an oblation to this paynim djinni we must create an instance of this deity to which we can tender our offering. This is done through a call to the follows function

```
GATAdvertService GATAdvertService_Create( GATContext context,
                                          GATPreferences_const preferences)
```

The first argument to this function is a `GATContext` instance, the details of which have been covered previously. The next argument is a `GATPreferences_const` instance, the details of which are covered in Appendix D. This function returns an instance of the class `GATAdvertService` corresponding to the data passed to it but returns `NULL` if it encounters any problems in creating such a deity.

An as example of this function in the act of djinni creation, let us consider this little code snippet

```
GATContext context;
GATAdvertService advertService;

context = ...

advertService = GATAdvertService_Create( context, NULL );
```

Now after wishing this djinni into existence and making it do any little dance we see fit, we need to put it down. Hey, no one mans the hallucinogen spilling fissures at the Delphi anymore or pays homage to Aeolus, Aether, or Aphrodite; so, why should we keep our djinni around any longer than needed. Well, the that light burns twice as bright burns half as long, and our djinni has burned so very, very brightly. Here is its Shiva

```
void GATAdvertService_Destroy(GATAdvertService *object)
```

The first argument to this function is a pointer to the `GATAdvertService` that one wishes to destroy. Upon return from this function all the resources which were tied up by the passed `GATAdvertService` instance are freed.

7.2.2 Adding Advertisables to an AdvertService

Now as we have mastered the art of creating and destroying `GATAdvertService` instances let us next move on to the task of adding advertisables to an `GATAdvertService` instance. This is done through a call to the following cunningly named function

```
GATResult
  GATAdvertService_Add( GATAdvertService object,
                        GATObject_const advertisable,
                        GATTable metadata,
                        GATString_const path)
```

The first argument to this function is a `GATAdvertService` instance. This is the instance to be modified. The next argument is a `GATObject` instance. This is the advertisable which is to added to the `GATAdvertService` and hence must be an instance of a class which implements the interface `GATInterface_IAdvertisable`. The next argument is a `GATTable`, a class covered in Appendix F. This `GATTable` instance contains the description of which we spoke earlier. This description takes the form of a set of name/value pairs in which the value is always a standard C string. For example, I might place an advertisable in a `GATAdvertService` instance with the name/value pair "owner/Leonardo" if I wanted to indicate that the particular instance I was placing was owned by Leonardo. Generically, one can use any C string for a key and any C string for a value. However, the keys beginning with `GAT_` are reserved for internal use by the GAT engine and the values may be what are called a "regular expressions." The reader not familiar with regular expressions should refer to Appendix E. The final argument to this function is a `GATString`, a class covered in Appendix G. Each entry in a `GATAdvertService` has associated with it a POSIX path, for those readers unfamiliar with the details of POSIX paths refer to Appendix I. This `GATString` instance contains a string representing the POSIX path which is to be associated with the advertisable passed as this function's second argument. Finally this function returns a `GATResult`, the details of which are covered in Appendix C, indicating its completion status.

As an example of this engine in action let us consider placing the a `GATFile` instnace in a `GATAdvertService`. This would look as follows

```
GATFile file;
GATResult result;
GATTable table;
GATString string;
GATObject object;
GATAdvertService advertService;

file = ...
table = ...
advertService = ...
string = GATString_Create( "/tmp/trash", 11, "ASCII" );

object = GATFile_ToGATObject( file );

result = GATAdvertService_Add( advertService, object, table, string );
```

```
if( GAT_SUCCEEDED( result ) )
{
  /* The file has been added to advertService with path string and meta-data table */
}
```

7.2.3 Deleting Advertisables from an AdvertService

Deleting an advertisable from a `GATAdvertService` is much easier than adding and advertisable to a `GATAdvertService` instance. One need only make use of the following function

```
GATResult GATAdvertService_Delete(GATAdvertService object, GATString_const path)
```

The first argument to this function is a `GATAdvertService` specifying the `GATAdvertService` instance from which avertisable are to be deleted. The second argument is a `GATString` instance which is the path of the advertisable to be deleted. This function returns a `GATResult`, covered in detail in Appendix C, which indicates the completion status of this function.

To take this machine out for a test drive lets consider a code snippet which could be used to delete the `GATFile` instance added in our previous example. The could snippet would have the following arc

```
GATResult result;
GATString string;
GATAdvertService advertService;

advertService = ...
string = GATString_Create( "/tmp/trash", 11, "ASCII" );

result = GATAdvertService_Delete( advertService, string );

if( GAT_SUCCEEDED( result ) )
{
  /* The advertisable at the path string has been removed */
}
```

7.2.4 Obtaining the Description of an Advertisable

After placing an advertisable in a `GATAdvertService` or having someone else place an advertisable in a `GATAdvertService` that you wish to examine, it's often the case that one needs to obtain a description of the advertiable at a givan POSIX path. This is accomplished through the function

```
GATResult GATAdvertService_GetMetaData( GATAdvertService_const object,
                                        GATString_const path,
                                        GATTable *metadata)
```

The first argument to this function is the `GATAdvertService` to be probed for information. The second argument is a `GATString` containing a POSIX path. This is the path of the advertisable we wish to obtain the description of. the final argument to this function is a pointer to a `GATTable`. Through this pointer the caller is returned the description, a `GATTable` instance, associated with the advertisable when it was placed in the `GATAdvertService`. This function returns a `GATResult`, a type covered in Appendix C, which indicates this function's completion status.

As an example consider the case in which on machine A some code has placed many advertisables into a `GATAdvertService` then given you, on machine B, the list of the POSIX paths of all these advertisables. You want to use some, but not all, of these various advertiables, but only one which have a description that jibes with you goals. So, to examine the description of these various advertiables you would need to write a function which, given a POSIX path, returns a description of the corresponding advertisable. The function might look a bit like the following

```
GATResult GetDescription( GATString_const path, GATTable *description )
{
  GATResult result;
  GATContext context;
  GATAdvertService advertService;

  result = GAT_MEMORYFAILURE;

  context = GATContext_Create();
  if( NULL != context )
  {
    advertService = GATAdvertService_Create( context, NULL );
    if( NULL != advertService )
    {
      result = GATAdvertService_GetMetaData( advertService, path, description );

      GATAdvertService_Destroy( &advertService );
    }
    GATContext_Destroy( &context );
  }

  return result;
}
```

7.2.5 Finding Advertisables

A common use case for the `GATAdvertService` is one in which an application searches in a `GATAdvertService` for an advertisable which fits a certain description. For example, one may try and search for all descriptions containing the key/value pair "owner/Leonardo". A search of this type is accomplished through us of the following function

```
GATResult GATAdvertService_Find(GATAdvertService_const object,
                                GATTable_const metadata,
                                GATList_String *paths)
```

The first argument to this function is the `GATAdvertService` whose contents are to searched. The second argument to this function is a `GATTable` the description of the advertisable the search will find. As one will recall, this is a set of key/value pairs in which the key is a standard C string not beginning with `GAT_` and the value is also a POSIX regular expression expressed as a standard C string[6]. The final argument to this function is a pointer to a `GATList_String`. It is through this pointer that the search's results are returned to the caller. The results are presented as a list of standard C strings in which each string contains a POSIX path to an advertisable matching the passed description. Finally this function returns a `GATResult`, covered in Appendix C, which indicates the completion status of this function.

As an example of this function in action, consider creating a command line tool which, given a description in the form of a set of key/value pairs, prints out a list of POSIX paths of the various advertisables which fit the passed description. Such a command line to might have within its code a function which, when passed a `GATTable` describing the desired advertisables, passes back a list of standrd C strings each containing the POSIX path of an advertisable which fits the passed description. Such a function might look like this

```
GATResult GetPaths( GATTable_const description, GATList_String *paths )
{
  GATResult result;
  GATContext context;
  GATAdvertService advertService;

  result = GAT_MEMORYFAILURE;

  context = GATContext_Create();
  if( NULL != context )
  {
    advertService = GATAdvertService_Create( context, NULL );
    if( NULL != advertService )
    {
      result = GATAdvertService_Find( advertService, description, paths );

      GATAdvertService_Destroy( &advertService );
    }
    GATContext_Destroy( &context );
  }

  return result;
}
```

7.2.6 Getting or Deleting an Advertisable

After obtaining the POSIX path of an advertisable in an `GATAdvertService` usually one would like to get the actual advertisable instance located at this POSIX path. This is done through the following function

```
GATResult GATAdvertService_GetAdvertisable( GATAdvertService_const object,
                                            GATString_const path,
                                            GATObject *advertisable)
```

The first argument to this function is the `GATAdvertService` from which the advertisable is to be gotten. The second argument to this function is a `GATString` instance containing the

[6] The reader unfamiliar with POSIX regular expressions is urged to refer to Appendix E.

POSIX path of the advertisable to extract. The final argument is a pointer to a `GATObject`. It is through this pointer that the `GATAdvertService` returns to the caller the advertisable located at the passed POSIX path. One should note that this call does not delete the advertiable from the `GATAdvertService`. It simply hands the caller a "clone" of the advertiable placed in the `GATAdvertService`. Finally, this function returns a `GATResult`, covered in Appendix C, which indicates the completion status of this function.

To take this function out for a test drive consider the case in which code on machine A has placed a `GATFile` into the `GATAdvertService` under some fixed POSIX path then through some means transfers this POSIX path to machine B which is supposed to get this advertiable from the `GATAdvertService`. Such an application on machine B would likely have a function of the following form

```
GATResult GetAdvertisable( GATString_const path, GATObject *advertisable )
{
  GATResult result;
  GATContext context;
  GATAdvertService advertService;

  result = GAT_MEMORYFAILURE;

  context = GATContext_Create();
  if( NULL != context )
  {
    advertService = GATAdvertService_Create( context, NULL );
    if( NULL != advertService )
    {
      result =
        GATAdvertService_GetAdvertisable( advertService, path, advertisable );

      GATAdvertService_Destroy( &advertService );
    }
    GATContext_Destroy( &context );
  }

  return result;
}
```

After getting an advertisable at a given path from the `GATAdvertService` it is often the case that one wishes to actually delete the corresponding advertisable from the `GATAdvertService`. As mentioned above, the act of getting the advertisable simply gives the caller a "clone" of the advertisable placed in the `GATAdvertService`. The actual advertisable, after such a "Get," still resides in the `GATAdvertService`. To delete such an advertisble from the `GATAdvertService` one uses the function

```
GATResult GATAdvertService_Delete(GATAdvertService as, GATString_const path)
```

The first argument to this function is the `GATAdvertService` from which the advertisable is to be removed. The second argument is a `GATString` which contains the POSIX path of the advertisable to be removed from the passed `GATAdvertService`. This function also returns a

GATResult, covered in Appendix C, which indicates the completion status of this function. Upon successful completion of this function the advertisable associated with the specified POSIX path will be removed from the GATAdvertService.

If we continue in the same vein as the previous example, we may, after getting the advertisable form the GATAdvertService, wish to delete the advertisable from the same GATAdvertService as no other processes will have need for such an advertisable. If the application on machine B were indeed to do so, then it might have a function of the following form to accomplish this goal

```
GATResult DeleteAdvertisable( GATString_const path )
{
  GATResult result;
  GATContext context;
  GATAdvertService advertService;

  result = GAT_MEMORYFAILURE;

  context = GATContext_Create();
  if( NULL != context )
  {
    advertService = GATAdvertService_Create( context, NULL );
    if( NULL != advertService )
    {
      result = GATAdvertService_Delete( advertService, path );

      GATAdvertService_Destroy( &advertService );
    }
    GATContext_Destroy( &context );
  }

  return result;
}
```

7.2.7 Setting and Getting the Working Directory

A point which we have, as up until this moment ignored is the fact that a POSIX path can be not only an absolute path but also also a relative path. What happens if one uses a relative POSIX path for something like the “Get” function above? Well, it works.

The GATAdvertService carries internal state in the form of a “working directory.” A working directory is a POSIX path which is used to resolve relative POSIX paths passed to the GATAdvertService. As an example , consider the case in which an GATAdvertService has the working directory

```
/Users/lenardo/unfinished/house/
```

then is asked to “Get” the advertisable at POSIX path

```
../ufizzi/waterpeople
```

The `GATAdvertService` would the resolve this relative POSIX path against the working directory and get the advertisable at the absolution POSIX path

```
/Users/lenardo/unfinished/ufizzi/waterpeople
```

A similar resolution process will occur for any `GATAdvertService` which accepts a POSIX path. So, you may ask, how does this working directory get set?

The working directory for a `GATAdvertService` get set through a call to the following function

```
GATResult GATAdvertService_SetPWD( GATAdvertService as, GATString_const path )
```

The first argument to this function is the `GATAdvertService` whose working directory is to be set. (One should note even through a `GATAdvertService` is accessible across machine boundaries the working directory of a `GATAdvertService` is a property which is specific to a given `GATAdvertService` instance. So, setting the working directory on instance A of a `GATAdvertService` has no effect upon the working directory of `GATAdvertService` instance B.) The next argument to this function is a `GATString` which is the POSIX path of the desired working directory. Again, this POSIX path can also be a relative path and would, in this case, be resolved against the current working directory of the `GATAdvertService`. Finally, this function returns a `GATResult` which indicates the completion status of this function.

As one may set the working the directory, the next natural thing one may wish to do is to get the working directory that one has just set or that, more commonly, has been set by some other code. This is achieved through use of the function

```
GATResult GATAdvertService_GetPWD(GATAdvertService object, GATString *path)
```

the first argument is the `GATAdvertService` one wishes to obtain the working directory of. The second argument is a pointer to a `GATString`. It is through this pointer that a `GATString` instance containing the working directory of the passed `GATAdvertService` is returned to the caller. Finally, this function returns a `GATResult`, a type covered in Appendix C, which indicates the completion status of this function.

7.3 Some Useful Programs

In this section we will continue in our vein of creating command line programs which hopefully will be of some use the the GAT application programer and user. However, as most operating systems don't have anything akin to a `GATAdvertService` lurking within their bowels, we'll have to, instead of aping common tools, make up some command line tools which seems as if they would be useful. Here we go.

7.3.1 What is Big, Red, and Eats Rocks?

This question has often vexed me since my childhood, "What's big, red, and eats rocks?" As anyone who grew up with the excellent children's book *Big Red Rock Eater* knows, a big red rock eater is big red and eats rocks. (Sometimes the most complicated questions have the simplest answers.) So what does this have to do with GAT much less the `GATAdvertService`?

We will next create a command line program, which we will christen `brre`, pronounced like "brie" and whose letters are an abbreviation for the phrase "Big Red Rock Eater." This program given an set of key/value pairs will print out the POSIX paths of all the various advertisables which fit the description created by the passed key/value pairs. For example, one could call `brre` as follows

```
% brre size=big color=red diet=rocks
```

The program would then respond with a list of POSIX paths which might look like this

```
/Users/MrBigRedRockEater
/Users/MrsBigRedRockEater
```

In addition these command line arguments can also contain POSIX regular expressions as values, see Appendix E for the details of what a POSIX regular expression is. For example, one might also use `brre` as follows

```
% brre size=big color=r* diet=rocks
```

which might yield the results

```
/Users/MrsBigRotRockEater
/Users/MrBigRedRockEater
/Users/MrsBigRedRockEater
```

The full code for the command line tool `brre` is as follows

```
#include <stdio.h>
#include <string.h>
#include "GAT.h"

int main( int argc, char *argv[] )
{
    int count;
    char *key;
    char *value;
    int returnValue;
    GATTable table;
    GATResult result;
    GATContext context;
    GATList_String strings;
    GATAdvertService advertService;
    GATList_String_Iterator currentIterator;

    /* Set result to a memory failure */
    result = GAT_MEMORYFAILURE;

    /* Create GATTable */
    table = GATTable_Create();
```

```
/* Check GATTable creation */
if( NULL != table )
{
  /* Loop over command line arguments */
  for( count = 1; count < argc; count++ )
  {
    /* Obtain key */
    key = strtok( argv[count], "=" );

    /* Obtain value */
    if( NULL != key )
      value = strtok( NULL, "=" );

    /* Place key and value in GATTable */
    if( (NULL != key) && (NULL != value) )
      result = GATTable_Add_String( table, key, value );

    /* Print out error */
     if( (NULL == key) || (NULL == value) || GAT_FAILED( result ) )
       printf( "Error parsing command line argument: %s\n", argv[count] );
  }

  /* Set result to a memory failure */
  result = GAT_MEMORYFAILURE;

  /* Create GATContext */
  context = GATContext_Create();

  /* Check GATContext creation */
  if( NULL != context )
  {
    /* Create GATAdvertService */
    advertService = GATAdvertService_Create( context, NULL );

    /* Check GATAdvertService creation */
    if( NULL != advertService )
    {
      /* Find Advertisables */
      result = GATAdvertService_Find( advertService, table, strings );

      /* Check Success of Last Call */
      if( GAT_SUCCEEDED( result ) )
      {
        /* Obtain Beginning GATList_String_Iterator */
        currentIterator = GATList_String_Begin( strings );

        /* Loop over strings */
        while( currentIterator != GATList_String_End( strings ) )
        {
```

```
          /* Print out POSIX path */
          printf( "%s\n", GATList_String_Get( currentIterator ) );

          /* Increment currentIterator */
          currentIterator = GATList_String_Next( currentIterator );
        }

        /* Destroy GATList_String */
        GATList_String_Destroy( &strings );
      }

      /* Destroy GATAdvertService */
      GATAdvertService_Destroy( &advertService );
    }

    /* Destroy GATContext */
    GATContext_Destroy( &context );
  }

  /* Destroy GATTable */
  GATTable_Destroy( &table );
}

/* Set returnValue and print error message */
returnValue = 0;
if( GAT_FAILED( result ) )
{
  returnValue = 1;

  printf( "There was an error in execution of brre\n" );
}

/* Return to OS */
return returnValue;
}
```

7.3.2 Who lives at 1600 Pennsylvania Avenue?

Who lives at 1600 Pennsylvania Avenue? It's often the case that one has the POSIX path of an advertisable in a `GATAdvertService` but yet has no information, short of the POSIX path itself, describing this advertisable. Our next command line program remedies this situation. In honour of the old **C64** command `peek` which would allow the application programmer to peek at the contents of memory we will call our new command line program `sonofpeek`.

Our new baby `sonofpeek` when presented with a POSIX path will print out all the key/value pairs describing the advertisable present at that POSIX path. For example, it could be called as follows

```
% sonofpeek /earth/usa/DC/1600PennsylvaniaAvenue
```

and then might respond with the following information

```
FirstName=George
MiddleName=Walker
LastName=Bush
Age=58
Height=183cm
Weight=87Kg
IQ=91
```

The full source for `sonofpeek` is as follows

```
#include <stdio.h>
#include <string.h>
#include "GAT.h"

int main( int argc, char *argv[] )
{
  int counter;
  char *keys[];
  int returnValue;
  GATTable table;
  GATResult result;
  GATString string;
  char value[2048];
  GATContext context;
  GATAdvertService advertService;

  /* Check command line arguments */
  if( 2 != argc )
  {
    /* Print out error message */
    printf( "usage: sonofpeek path\n" );

    /* Return to OS */
    return 1;
  }

   /* Set result to a memory failure */
   result = GAT_MEMORYFAILURE;

   /* Create GATContext */
   context = GATContext_Create();

   /* Check GATContext creation */
   if( NULL != context )
   {
     /* Create GATAdvertService */
     advertService = GATAdvertService_Create( context, NULL );
```

```
/* Check GATAdvertService creation */
if( NULL != advertService )
{
  /* Create GATString */
  string = GATString_Create( argv[1],
                             (GATuint32) (strlen( argv[1] ) + 1),
                             "ASCII" );

  /* Check GATString creation */
  if( NULL != string )
  {
    /* Get description */
    result = GATAdvertService_GetMetaData( advertService, string, &table );

    /* Check success of GATAdvertService_GetMetaData */
    if( GAT_SUCCEEDED( result ) )
    {
      /* Get keys */
      keys = (char **) GATTable_GetKeys( table );

      /* Loop over keys */
      counter = 0;
      while( NULL != keys[counter] )
      {
        /* Get value */
        result = GATTable_Get_String( table, keys[counter], value, 2048 );

        /* Check success of GATTable_Get_String */
        if( GAT_SUCCEEDED( result ) )
        {
          /* Print key/value pair */
          printf( "%s=%s\n", keys[counter], value );
        }
        else
        {
          /* Print error */
          printf( "Error obtaining value for key %s\n", keys[counter] );
        }

        /* Increment counter */
        counter = counter + 1;
      }

      /* Destroy keys */
      GATTable_ReleaseKeys( table, keys );
    }

    /* Destroy GATString */
    GATString_Destroy( &string );
```

```
      }

      /* Destroy GATAdvertService */
      GATAdvertService_Destroy( &advertService );
    }

    /* Destroy GATContext */
    GATContext_Destroy( &context );
  }

  /* Set returnValue and print error message */
  returnValue = 0;
  if( GAT_FAILED( result ) )
  {
    returnValue = 1;

    printf( "There was an error in execution of sonofpeek\n" );
  }

  /* Return to OS */
  return returnValue;
}
```

8 Resource Management

8.1 The Resource Management Package

Up until this instant the Grid, to the hapless application programmer, is nothing more than the little cloud seen so often in useless internet diagrams, for example figure 20.

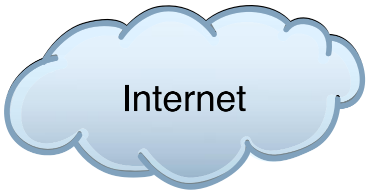

Figure 20: A useless diagram of the internet.

This is at once good and bad; let me explain. The application programmer has many things to worry about: Did their kid get to school OK? Do the users really need this extra feature that will take a year to implement and only be used once? Is my stock portfolio sufficiently diversified?... Sometimes they don't want to be bothered with the details of the all the various protocols and computers that together constitute the "grid." They simply want to use grid and be done with it. For these uses, this abstract picture of the grid is just the ticket.

However, there are other cases when the details actually do matter. For example, consider the case in which you need to move a large file off of one computer, as the drive is running low on space. You would need to know something about the target computer; in particular, you would need to know how much disk space it has. As another example, consider if you were trying to run a program which required a lot of CPU power on a remote computer; you couldn't run it on the local machine, it's to wimpy. You'd need to find something about the remote machine, for example the number of CPU's, the type of CPU's, the amount of memory ... In short you'd actually need to take a magnifying glass to this "grid" to see what's really there, for example figure 21.

From Chapter 3 on we've had no need of such a magnifying glass, the cloud of the grid has served us well through these chapters. However, this blissful ignorance has its limits and can only bring us so far. Eventually we will need this magnifying glass. So, in this chapter we will learn to use the magnifying glass of GAT, the resource management package.

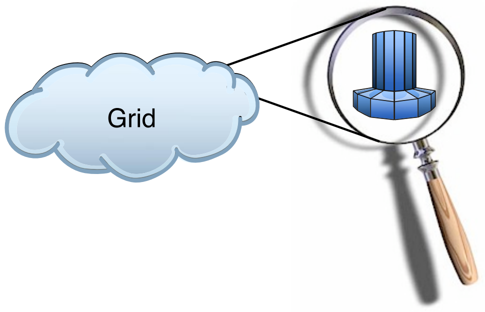

Figure 21: A details of a useless diagram of the grid.

8.2 Finding Resources

The resource management package allows the application programmer to finger the finer details of the gird, affording them a view at every nook and cranny constituting this "grid." In particular it allows the application programmer to train her magnifying glass on any portion of the gird: networks, CPU's, software... and examine, at close detail, the face these components present to the world.

The application programmer need only describe where she wishes to train this magnifying glass, "Show me all the Linux boxes with more than 2 Gigs of memory," and the minions of the resource management package comply, giving the application programmer the details of every resource that matches the query. Beyond simply finding hardware resources the resource management package also can find, for the application programmer, software resources. For example, an application programmer may be working with a piece of software which requires a plugin of type X to function. So, the application programmer may query the resource management system for the presence of such a plugin, "Show me all the plugins of type X", and the minions of GAT will comply.

8.3 Making Plans and Changing Your Mind

Usually when you find something, such as a hotel room, you need not only find it, but actually reserve it so you can use it. What good would it be to find that there exists a free room in the Gallery Art Hotel only a hop-step and a jump from the Pointe Vecchio in Florence, when you can't reserve it? The same is true of GAT resources. What if, like above, you found that there were 10 Linux boxes with more than 2 Gigs of memory. Then what? If you couldn't somehow reserve these resources, then by the time you got around to actually trying to use them some other early bird might have gotten your worm. So, you can see that the ability to reserve resources is extremely important, and of course GAT gives you that ability and more.

GAT also allows you to change your mind. Say if you had just finished writing the code for a simulation of two inspiralling black holes so accurate that your reams of code are threatening to collapse into their own event horizon, then you reserve a mare virile enough to ride the burden of your code all the way to coalescence; however, there's many a slip 'twixt the cup and the lip. After making the reservation you realize, to your horror, that an "off-by-one error" was introduced in to your most inner of loops. Oops! Now you don't really need this prize mare to be waiting around to dispatch the magic that was your code. You need to cut her loose. GAT allows you to do just this, reserve a resource, then at a later date, cancel this reservation. GAT, manna of the grid application programmer.

8.4 The Resource Management Package

The resource management package has three main constituents "resource descriptions," "resources," and a "resource broker."

8.4.1 Resource Descriptions

A resource description is, like the name suggests, a description of a resource. There are two types of resource descriptions the class `GATSoftwareResourceDescription`, which is used to describe software resources, and the class `GATHardwareResourceDescription`, used to describe hardware resources. These classes both extend the abstract class `GATResourceDescription`. The situation is that depicted in figure 22.

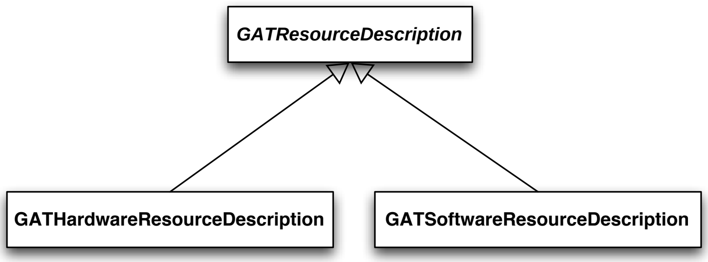

Figure 22: The various GATResourceDescription classes.

8.4.2 Resources

A resource is any hardware or software entity providing some capability, i.e. a hardware or software resource. All classes representing resources are tagged as such through the interface `GATInterface_IResource`, which they all realize. In particular, there exist two classes which realize this interface `GATSoftwareResource`, a class which represents a software resource, and `GATHardwareResource`, a class which, as the name implies, represents a hardware resource. For example, an instance of the class `GATSoftwareResource` might represent a program to simulate the weather in Tokyo on the June 6th in the year 2020 while an instance of the class `GATHardwareResource` might represent the hardware on which to run this code, the Earth Simulator for example.

8.4.3 Resource Broker

Finally, the resource broker is agent whose responsibility is to find or reserve resources for the application programmer. The resource broker is represented in GAT through the class

`GATResourceBroker`, an instance of which is used to find or reserve resources for the application programmer. Given a resource description the resource broker will proffer up all the known resources, to which the application has access rights, thus, finding resources based only on their description. In addition, the resource broker, given a resource description, can reserve a suitable resource for a user configurable time. The resource broker can also play this same tune when given a resource, not simply a resource description, allowing the application programmer to reserve specific resources.

8.4.4 Constructing and Destroying HardwareDescription Instances

Now, as the turbid, conceptual backwaters of the resource management package are, hopefully, a little less murky, we can move on to the process of putting all these abstractions into practice. We will begin this process by examining the creation and destruction of a `GATHardwareResourceDescription`.

The class `GATHardwareResourceDescription` can most simply be conceptualized as a container for a `GATTable`, a class covered in Appendix F. So, it's a small wonder that to create a `GATHardware Description` instance one uses a function of the following form

```
GATHardwareResourceDescription
 GATHardwareResourceDescription_Create(GATTable_const attributes)
```

The first argument to this function is a `GATTable` instance. This `GATTable` instance, the contents of which we will cover below, contains the description of the hardware resource we are trying to represent. This function returns a `GATHardwareResourceDescription` or `NULL` upon the occurrence of an error.

A hardware resource, at least as far as GAT is concerned, can be described by a set of name/value pairs. For example, one could specify that a hardware resource has a PowerPC CPU through the following name/value pair

```
cpu.type=powerpc
```

One similarly could specify that a hardware resource be a Power Macintosh by using the next name/value pair

```
machine.type=Power Macintosh
```

To construct a `GATHardwareResourceDescription`, as we saw above, one requires a `GATTable` instance. It is this `GATTable` instance which contains the various name/value pairs describing the resultant `GATHardwareResourceDescription`.

So, you may ask, can I place any name/value pairs in such a `GATTable`. Well, the answer is yes; however, GAT is only required to support a certain set of name value pairs. The extra name/value pairs may be ignored by GAT. The full set of supported name/value pairs, along with each value's type, can be found in table 5.

If a particular name/value pair is not specified in a particular `GATTable` instance, then GAT assumes that this name/value pair can take on any value. So, for example, if the there is no key

Name	Type	Description
`memory.size`	`Float`	The minimum memory in GB.
`memory.accesstime`	`Float`	The minimum memory access time in ns.
`memory.str`	`Float`	The minimum sustained transfer rate in GB/s.
`machine.type`	`String`	The machine type as returned from `uname -m`.
`machine.node`	`String`	The machine node name as returned from `uname -n`.
`cpu.type`	`String`	The generic cpu type as returned from `uname -p`.
`cpu.speed`	`Float`	The minimum cpu speed in GHz.
`disk.size`	`Float`	The minimum size of the hard drive in GB.
`disk.accesstime`	`Float`	The minimum disk access time in ms.
`disk.str`	`Float`	The minimum sustained transfer rate in MB/s.

Table 5: Hardware Resource Description: The minimum set of supported name/values.

`disk.accesstime` in a `GATTable` instance, then GAT assumes that this property can take on any value. In addition, if one specifies a particular name/value pair, say `memory.size=1024`, GAT will make the obvious assumption that this specification would describe a hardware resource with 1024 or more GB of memory.

To destroy the so created `GATHardwareResourceDescription` one uses the following function

```
void
  GATHardwareResourceDescription_Destroy(GATHardwareResourceDescription *resource)
```

This function takes as its first argument a pointer to a `GATHardwareResourceDescription`. This points to the `GATHardwareResourceDescription` to be destroyed. Upon successful completion this any resources this `GATHardwareResourceDescription` maintained will be released.

8.4.5 Constructing and Destroying SoftwareResourceDescription Instances

Constructing and destroying a `GATSoftwareResourceDescription` is similar to creating and destroying a `GATHardwareResourceDescription` instance. One uses the following function

```
GATSoftwareResourceDescription GATSoftwareResourceDescription_Create(
                                                    GATTable_const attributes)
```

This function takes a `GATTable` instance and returns a `GATSoftwareResourceDescription` upon success, upon failure it returns `NULL`. The passed `GATTable` contains a description, i.e. various name/value pairs, of the returned `GATSoftwareResourceDescription`. The supported name/value pairs which can occur in this `GATTable` instance are given in table 6

Again, if a particular name/value pair is not specified in a particular `GATTable` instance, then GAT assumes that this name/value pair can take on any value. So, for example, if the there is no key `os.release` in a `GATTable` instance, then GAT assumes that this property can take on any value.

To destroy the so created `GATSoftwareResourceDescription` one uses the following function

Name	Type	Description
`os.name`	`String`	The os name as returned from `uname -s`.
`os.type`	`String`	The os type as returned from `uname -p`.
`os.version`	`String`	The os version as returned from `uname -v`.
`os.release`	`String`	The os release as returned from `uname -r`.
`os.name`	`String`	The os name as returned from `uname -s`.

Table 6: Software Resource Description: the minimum set of supported name/values.

```
void
  GATSoftwareResourceDescription_Destroy(GATSoftwareResourceDescription *resource)
```

This function takes as its first argument a pointer to a `GATSoftwareResourceDescription`. This points to the `GATSoftwareResourceDescription` to be destroyed. Upon successful completion this any resources this `GATSoftwareResourceDescription` maintained will be released.

8.4.6 Constructing and Destroying ResourceBroker Instances

As described previously, to actually make use of a `GATHardwareResourceDescription` or `GATSoftware Description` to find or reserve resource one needs to first create a `GATResourceBroker`. So, before we can train our magnifying glass upon the grid we need to first understand how to create. and subsequently destroy, `GATResourceBroker` instances.

To create a `GATResourceBroker` instance one employs the function

```
GATResourceBroker
  GATResourceBroker_Create(GATContext context, GATPreferences_const preferences,
    GATString vo_name)
```

The first argument to this function is a `GATContext`, the details of which we have previously covered. The next argument is a `GATPreferences` instance, the details of which are covered in Appendix D. The final argument to this function is a `GATString`. This `GATString` identifies the so called "virtual organization" from which this resultant `GATResourceBroker` can cull resources.

A virtual organization[7] is a "dynamic collections of individuals, institutions, and resources." So, what exactly does that mean? Let's illustrate with an example. A particular organization, for example A.P.E., the **A**gency to **P**revent **E**vil, would be considered a virtual organization. In addition coalitions of organizations could also be considered as a virtual organization; for example, the coalition of institutions which work together as part of the GridLab project are a virtual organization. In general any organization or umbrella organization is a virtual organization.

The format which the `GATString` instance that identifies the virtual organization from which this resultant `GATResourceBroker` can cull resources is as of yet ill defined. Currently there is no standard format which the name of a virtual organization takes. Hence, to maintain maximal flexibility, GAT allows for the application programmer to specify this virtual organization with

[7] Foster, Kesselman, and Tuecke, *The Anatomy of the Grid*, Intl J. Supercomputer Applications, 2001.

an arbitrary GATString instance.

Finally the function GATResourceBroker_Create returns a GATResourceBroker corresponding to the passed information. However, it returns NULL if there is a problem in creating the specified GATResourceBroker instance.

As an example of this function is action let us consider the creation of a GATResourceBroker instance for the virtual organization "www.gridlab.org" A code snippet which would create such a GATResourceBroker would look as follows

```
GATString string;
GATContext context;
GATResourceBroker resourceBroker;

context = GATContext_Create();

if( NULL != context )
{
  string = GATString_Create( "www.gridlab.org", 16, "ASCII" );

  if( NULL != string )
  {
    resourceBroker = GATResourceBroker_Create( context, NULL, string );

    if( NULL != resourceBroker )
    {
      /* Do something! */
    }

    GATString_Destroy( &string );
  }

  GATContext_Destroy( &context );
}
```

To destroy such a GATResourceBroker instance and clean up any resources tied up by such an instance one calls the function

```
void GATResourceBroker_Destroy(GATResourceBroker *resource)
```

This function takes as its first, and only, argument a pointer to a GATResourceBroker. This points to the GATResourceBroker instance to be destroyed. For example, we can extend the previous code snippet to clean up the GATResourceBroker instance as follows

```
GATString string;
GATContext context;
GATResourceBroker resourceBroker;

context = GATContext_Create();
```

```
if( NULL != context )
{
  string = GATString_Create( "www.gridlab.org", 16, "ASCII" );

  if( NULL != string )
  {
    resourceBroker = GATResourceBroker_Create( context, NULL, string );

    if( NULL != resourceBroker )
    {
      /* Do something! */

      GATResourceBroker_Destroy( &resourceBroker );
    }

    GATString_Destroy( &string );
  }

  GATContext_Destroy( &context );
}
```

8.4.7 Finding Resources

As we now have constructed the foundation required to build anything of use with the resource management package, we can move on to the process of actually building on this foundation. The first brick that we will lay in this vein will teach us how to find resources fitting a particular resource description.

To find resources depicted through a particular resource description we use the following function

```
GATResult
  GATResourceBroker_FindResources(GATResourceBroker broker,
    GATResourceDescription_const description, GATList_GATResource *resources)
```

The initial argument to this function is a `GATResourceBroker` identifying the `GATResourceBroker` which is going to be used to carry-out the search. The second argument is a `GATResource Description` which describes the resource(s) to be found. The final argument is a pointer to a list of `GATResource` instances. It is through this pointer that this call returns to the caller a list of the resources found. Finally this function returns a `GATResult`, covered in Appendix C, which indicates the completion status of this function.

As a quick example, we can consider using this function to create a function that returns a list of all hardware resources a within a specified virtual organization. A code snippet which works this magic is as follows

```
GATResult GiveMeItAll( GATString virtualOrg )
{
  GATResult result;
  GATTable table;
  GATContext context;
```

```
  GATList_GATResource resources;
  GATResourceBroker resourceBroker;
  GATHardwareResourceDescription hardwareResourceDescription;

  result = GAT_MEMORYFAILURE;

  context = GATContext_Create();
  if( NULL != context )
  {
    resourceBroker = GATResourceBroker_Create( context, NULL, virtualOrg );
    if( NULL != resourceBroker )
    {
      table = GATTable_Create();
      if( NULL != table )
      {
        hardwareResourceDescription = GATHardwareResourceDescription_Create( table );
        if( NULL != hardwareResourceDescription )
        {
          result =
            GATResourceBroker_FindResources( resourceBroker,
                                             hardwareResourceDescription,
                                             &resources );
          if( GAT_SUCCEEDED( result ) )
          {
            /* Do something with resources! */

            GATList_GATResource_Destroy( &resources );
          }

          GATHardwareResourceDescription_Destroy( &hardwareResourceDescription );
        }

        GATTable_Destroy( &table );
      }

      GATResourceBroker_Destroy( &resourceBroker );
    }

    GATContext_Destroy( &context );
  }

  return result;
}
```

8.4.8 Reserving Resources

GAT allows for two different resource reservation methods. The first method describes the desired resource through the use of a `GATResourceDescription` instance while the second uses an `GATResource` instance to the same effect. We will cover both methods; let us begin by examining resource reservation through the use of a `GATResourceDescription` instance.

Reserving Resources Using a Resource Description: GAT provides the following call to allow for resource reservation using a `GATResourceDescription` instance

```
GATResult
  GATResourceBroker_ReserveResource_Description(GATResourceBroker broker,
    GATResourceDescription_const description, GATTime_const time,
    GATTimePeriod_const duration, GATReservation *reservation)
```

The initial argument to this function is the `GATResourceBroker` instance which is used to call in the reservation. The next argument is a `GATResourceDescription` instance used to describe the the desired resource. The following argument is a `GATTime` instances, the details of which are covered in Chapter 3, which indicates the time at which this reservation should start. The very next argument is a `GATTimePeriod`, a class which we will cover below, that signal the duration of this reservation. For example, a reservation may start on 00:00 GMT June 6 2020 and last for 24 hours.

This class `GATTimePeriod`, as one may gather from above, specifies a period of time. One creates an instance of such a class using the function

```
GATTimePeriod GATTimePeriod_Create(GATdouble64 duration)
```

This function will, upon success, return a `GATTimePeriod` instance which represents a period of time `duration` seconds long. If this call fails it returns `NULL`. One can also create an instance using the function

```
GATTimePeriod GATTimePeriod_Create_Difference(GATTime start, GATTime end)
```

which upon success creates a `GATTimePeriod` that starts at `start` and ends at `end`. One destroys such an instance using the function

```
void GATTimePeriod_Destroy(GATTimePeriod *resource)
```

which releases all resource held by the passed `GATTimePeriod` instance. Finally one can examine the duration represented by a `GATTimePeriod` instance through a call to the function

```
GATdouble64 GATTimePeriod_GetDuration(GATTimePeriod_const period
```

which returns the duration in seconds represented by the `GATTimePeriod` instance `period`.

The final argument to the function `GATResourceBroker_ReserveResource_Description` is a pointer to a `GATReservation`. It is through this pointer that the function returns to the caller an instance of the class `GATReservation` which represents a reservation.

The application programmer can not directly create an instance of the class `GATReservation`. An application programmer can only obtain such instances through calls of the above ilk. The application programmer is, however, responsible for destroying such `GATReservation` instances. This is done through a call to the function

```
void GATReservation_Destroy(GATReservation *resource)
```

which releases any resources held be the passed `GATReservation` instance. After obtaining such a `GATReservation` instance one can obtain the `GATResource` to which this `GATReservation` corresponds through the call

```
GATResult GATReservation_GetResource(GATReservation_const reservation,
                                     GATResource_const *resource)
```

Finally we complete our study of the original function `GATResourceBroker_ReserveResource_Description` by noting that it returns a `GATResult`, covered in Appendix C, which indicates the completion status of the function.

Reserving Resources Using a Resource: The second manner in which GAT allows resource to be reserved is through the use of an actual `GATResource` instance. One may, for example, obtain a `GATResource` instance through a call to the "Find Resources" call, then wish to actually make a reservation on that resource. The following function

```
GATResult
  GATResourceBroker_ReserveResource(GATResourceBroker broker,
    GATResource_const resource, GATTime_const zeit,
    GATTimePeriod_const duration, GATReservation *reservation)
```

makes this possible. The semantics of this call are exactly the same as the previous "Reserve" function mod the obvious replacement of `GATResourceDescription` with `GATResource`.

8.4.9 Canceling Reservations

After one has a `GATReservation` instance, canceling the reservation corresponding to this instance is relatively simple. One simply call the function

```
GATResult GATReservation_Cancel(GATReservation reservation)
```

Which takes as its initial argument the `GATReservation` instance to be canceled. The function returns a `GATResult`, covered in Appendix C, which indicates the completion status of the function.

8.5 Some Useful Programs

8.5.1 Find me something big and fast!

As an example of the ground covered in this chapter, we now introduce a command line utility which we'll paint with the epithet `hwloupe`, the prefix `hw` standing for hardware and the suffix `loupe` representing itself, a small magnifying glass.

This command line utility will when presented with a set of name/value pairs

```
% hwloupe virtualorg memory.size=1024 cpu.type=powerpc
```

will create a `GATHardwareResourceDescription` corresponding to this set of name value pairs, then find all resources which are described by this `GATHardwareResourceDescription` in the virtual organization `virtualorg`. Upon doing so it will print out its finding in the following format

```
memory.size=1024
memory.accesstime=10
memory.str=100
machine.type=Power Macintosh
machine.node=L-DaVinci1s-Computer.local
cpu.type=powerpc
cpu.speed=1
disk.size=100
disk.acesstime=4
disk.str=500

memory.size=2048
memory.accesstime=5
memory.str=1000
machine.type=Power Macintosh
machine.node=L-DaVinci2s-Computer.local
cpu.type=powerpc
cpu.speed=10
disk.size=1000
disk.acesstime=4
disk.str=5000

...
```

Here's the full source of `hwloupe`

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "GAT.h"

static void printGATList_GATResource( GATList_GATResource resources );
static GATResult addNameValue( GATTable table, const char *name, const char *value );

int main(int argc, char *argv[])
{
  int counter;
  char *name;
  char *value;
  int returnValue;
  GATTable table;
  GATResult result;
  GATString string;
  GATContext context;
  GATList_GATResource resources;
```

```
GATResourceBroker resourceBroker;
GATHardwareResourceDescription hardwareResDes

/* Check command line arguments */
if( argc < 2 )
{
  /* Print out error message */
  printf( "Usage: hwloupe virtualorg [name=value]*\n" );

  /* Return to OS */
  return 1;
}

/* Set result to a memory failure */
result = GAT_MEMORYFAILURE;

/* Create GATContext */
context = GATContext_Create( );

/* Check GATContext creation */
if( NULL != context )
{
  /* Create GATString */
  string = GATString_Create( argv[1], strlen( argv[1] ) + 1, "ASCII" );

  /* Check GATString creation */
  if( NULL != string )
  {
    /* Create GATResourceBroker */
    resourceBroker = GATResourceBroker_Create( context, NULL, string );

    /* Check GATResourceBroker creation */
    if( NULL != resourceBroker )
    {
      /* Create GATTable */
      table = GATTable_Create();

      /* Check GATTable creation */
      if( NULL != GATTable)
      {
        /* Loop over command line arguments */
        for( count = 2; count < argc; count++ )
        {
          /* Obtain name */
          name = strtok( argv[count], "=" );

          /* Obtain value */
          if( NULL != name )
            value = strtok( NULL, "=" );
```

```
        /* Place name and value in GATTable */
        result = addNameValue( table, name, value );

        /* Print out addition error */
        if( GAT_FAILED( result ) )
          printf( "Error in adding the name/value pair: %s/%s\n", name, value );
      }

      /* Set result to a memory failure */
      result = GAT_MEMORYFAILURE;

      /* Create GATHardwareResourceDescription *
      hardwareResDes =
       GATHardwareResourceDescription_Create( table );

      /* Check GATHardwareResourceDescription creation */
      if( NULL != hardwareResDes )
      {
        /* Find Resources */
        result =
         GATResourceBroker_FindResources( resourceBroker,
                                          hardwareResDes,
                                          &resources );

        /* Check success of last call */
        if( GAT_SUCCEEDED( result ) )
        {
          /* Print resources */
          printGATList_GATResource( resources );

          /* Destroy GATList_GATResource */
          GATList_GATResource_Destroy( &resources );
        }

        /* Destroy GATHardwareResourceDescription */
        GATHardwareResourceDescription_Destroy( &hardwareResDes );
      }

      /* Destroy GATTable */
      GATTable_Destroy( &table );
    }

    /* Destroy GATResourceBroker */
    GATResourceBroker_Destroy( &resourceBroker );
  }

  /* Destroy GATString */
  GATString_Destroy( &string );
```

```
    }

    /* Destroy GATContext */
    GATContext_Destroy( &context );
  }

  /* Return to OS */
  return returnValue;
}

static GATResult addNameValue( GATTable table, const char *name, const char *value )
{
  int added;
  GATResult result;
  GATfloat32 floatValue;

  /* Set result to an invalid parameter */
  result = GAT_INVALID_PARAMETER;

  /* Check parameters */
  if( (NULL != table) && (NULL != name) & (NULL != value) )
  {
    /* Set boolean */
    added = 0;

    /* Check if name is "memory.size" */
    if( 0 == strcmp( name, "memory.size" ) )
    {
      /* Convert value into GATfloat32 */
      floatValue = (GATfloat32) atof( value );

      result = GATTable_Add_float( table, (const void *) name, floatValue );
     }

     /* Check if name is "memory.accesstime" */
     if( 0 == strcmp( name, "memory.accesstime" ) )
     {
       /* Convert value into GATfloat32 */
       floatValue = (GATfloat32) atof( value );

      /* Add name/value pair */
       result = GATTable_Add_float( table, (const void *) name, floatValue );

       /* Flip boolean */
       added = 1;
     }

     /* Check if name is "memory.str" */
     if( 0 == strcmp( name, "memory.str" ) )
```

```
{
  /* Convert value into GATfloat32 */
  floatValue = (GATfloat32) atof( value );

 /* Add name/value pair */
  result = GATTable_Add_float( table, (const void *) name, floatValue );

  /* Flip boolean */
  added = 1;
}

/* Check if name is "cpu.speed" */
if( 0 == strcmp( name, "cpu.speed" ) )
{
  /* Convert value into GATfloat32 */
  floatValue = (GATfloat32) atof( value );

 /* Add name/value pair */
  result = GATTable_Add_float( table, (const void *) name, floatValue );

  /* Flip boolean */
  added = 1;
}

/* Check if name is "disk.size" */
if( 0 == strcmp( name, "disk.size" ) )
{
  /* Convert value into GATfloat32 */
  floatValue = (GATfloat32) atof( value );

  /* Add name/value pair */
  result = GATTable_Add_float( table, (const void *) name, floatValue );

  /* Flip boolean */
  added = 1;
}

/* Check if name is "disk.accesstime" */
if( 0 == strcmp( name, "disk.accesstime" ) )
{
  /* Convert value into GATfloat32 */
  floatValue = (GATfloat32) atof( value );

  /* Add name/value pair */
  result = GATTable_Add_float( table, (const void *) name, floatValue );

  /* Flip boolean */
  added = 1;
}
```

```
    /* Check if name is "disk.str" */
    if( 0 == strcmp( name, "disk.str" ) )
    {
      /* Convert value into GATfloat32 */
      floatValue = (GATfloat32) atof( value );

      /* Add name/value pair */
      result = GATTable_Add_float( table, (const void *) name, floatValue );

      /* Flip boolean */
      added = 1;
    }

    /* All others add as string values */
    if( 0 == added )
      result = GATTable_Add_String( table, (const void *) name, value );
  }

  /* Return to caller */
  return result;
}

static void printGATList_GATResource( GATList_GATResource resources )
{
  int counter;
  void **keys;
  float nextFloat;
  GATResult result;
  GATType nextType;
  char nextString[2048];
  GATTable_const table;
  GATResource resource;
  GATList_GATResource_Iterator end;
  GATList_GATResource_Iterator current;
  GATHardwareResource hardwareResource;
  GATHardwareResourceDescription hardwareResDes;

  /* Check parameters */
  if( NULL != resources )
  {
    /* Obtain Beginning Iterator */
    current = GATList_GATResource_Begin( resources );

    /* Check last call */
    if( NULL != current )
    {
      /* Obtain Ending Iterator */
      end = GATList_GATResource_End( resources );
```

```
/* Check last call */
if( NULL != end )
{
  /* Loop over resources */
  while( (NULL != current) && (current != end) )
  {
    /* Obtain next GATResource */
    resource = GATList_GATResource_Get( current );

    /* Check last call */
    if( NULL != resource )
    {
        /* Convert the GATResource to a GATHardwareResource
        hardwareResource = GATResource_ToGATHardwareResource( resource );

        /* Obtain the GATHardwareResourceDescription */
        result =
         GATHardwareResource_GetResourceDescription( hardwareResource,
                                                     &hardwareResDes );

        /* Check last call */
        if( GAT_SUCCEEDED( result ) )
        {
          /* Obtain GATTable */
          table =
            GATHardwareResourceDescription_GetDescription( hardwareResDes );

          /* Check last call */
          if( NULL != table )
          {
            /* Obtain keys */
            keys = GATTable_GetKeys( table );

            /* Check last call */
            if( NULL != keys )
            {
              /* Set counter */
              counter = 0;

              /* Loop over keys */
              while( NULL != keys[counter] )
              {
                /* Obtain value's type */
                nextType = GATTable_Get_ElementType( table, keys[counter]);

                /* Check type */
                if( GATfloat32 == nextType )
                {
```

```
                /* Obtain float */
                result =
                 GATTable_Get_float( table, keys[counter], &nextFloat );

                /* Check last call */
                if( GAT_SUCCEEDED( result ) )
                {
                  /* Print name=value */
                  printf( "%s=%f\n", (char *) keys[counter], nextFloat );
                }
              }

              /* Check type */
              if( GATType_String == nextType )
              {
                /* Obtain string */
                result =
                 GATTable_Get_String( table, keys[counter],
                                      nextString, 2048 );

                /* Check last call */
                if( GAT_SUCCEEDED( result ) )
                {
                  /* Print name=value */
                  printf( "%s=%s\n", (char *) keys[counter], nextString );
                }
              }

              /* Increament counter */
              counter = counter + 1;
            }

            /* Print blank line to separate resources */
            printf( "\n" );

            /* Destroy keys */
            GATTable_ReleaseKeys( table, keys );
          }
        }

        /* Destroy GATHardwareResourceDescription */
        GATHardwareResourceDescription_Destroy( &hardwareResDes );
      }
    }

    /* Obtain Next Iterator */
    current = GATList_GATResource_Next( current );
  }
}
```

```
    }
  }
}
```

8.5.2 An Equal Opportunity Employer

To not leave the software side out in the cold here's `swlopue`

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "GAT.h"

static void printGATList_GATResource( GATList_GATResource resources );
static GATResult addNameValue( GATTable table, const char *name, const char *value );

int main(int argc, char *argv[])
{
  int counter;
  char *name;
  char *value;
  int returnValue;
  GATTable table;
  GATResult result;
  GATString string;
  GATContext context;
  GATList_GATResource resources;
  GATResourceBroker resourceBroker;
  GATSoftwareResourceDescription softwareResDes

  /* Check command line arguments */
  if( argc < 2 )
  {
    /* Print out error message */
    printf( "Usage: hwloupe virtualorg [name=value]*\n" );

    /* Return to OS */
    return 1;
  }

  /* Set result to a memory failure */
  result = GAT_MEMORYFAILURE;

  /* Create GATContext */
  context = GATContext_Create( );

  /* Check GATContext creation */
  if( NULL != context )
  {
```

```
/* Create GATString */
string = GATString_Create( argv[1], strlen( argv[1] ) + 1, "ASCII" );

/* Check GATString creation */
if( NULL != string )
{
  /* Create GATResourceBroker */
  resourceBroker = GATResourceBroker_Create( context, NULL, string );

  /* Check GATResourceBroker creation */
  if( NULL != resourceBroker )
  {
    /* Create GATTable */
    table = GATTable_Create();

    /* Check GATTable creation */
    if( NULL != GATTable)
    {
      /* Loop over command line arguments */
      for( count = 2; count < argc; count++ )
      {
        /* Obtain name */
        name = strtok( argv[count], "=" );

        /* Obtain value */
        if( NULL != name )
          value = strtok( NULL, "=" );

        /* Place name and value in GATTable */
        result = addNameValue( table, name, value );

        /* Print out addition error */
        if( GAT_FAILED( result ) )
          printf( "Error in adding the name/value pair: %s/%s\n", name, value );
      }

      /* Set result to a memory failure */
      result = GAT_MEMORYFAILURE;

      /* Create GATSoftwareResourceDescription *
      softwareResDes =
       GATSoftwareResourceDescription_Create( table );

      /* Check GATSoftwareResourceDescription creation */
      if( NULL != softwareResDes )
      {
        /* Find Resources */
        result =
         GATResourceBroker_FindResources( resourceBroker,
```

```
                                                    softwareResDes,
                                                    &resources );

            /* Check success of last call */
            if( GAT_SUCCEEDED( result ) )
            {
              /* Print resources */
              printGATList_GATResource( resources );

              /* Destroy GATList_GATResource */
              GATList_GATResource_Destroy( &resources );
            }

            /* Destroy GATSoftwareResourceDescription */
            GATSoftwareResourceDescription_Destroy( &softwareResDes );
          }

          /* Destroy GATTable */
          GATTable_Destroy( &table );
        }

        /* Destroy GATResourceBroker */
        GATResourceBroker_Destroy( &resourceBroker );
      }

      /* Destroy GATString */
      GATString_Destroy( &string );
    }

    /* Destroy GATContext */
    GATContext_Destroy( &context );
  }

  /* Return to OS */
  return returnValue;
}

static GATResult addNameValue( GATTable table, const char *name, const char *value )
{
  GATResult result;
  GATfloat32 floatValue;

  /* Set result to an invalid parameter */
  result = GAT_INVALID_PARAMETER;

  /* Check parameters */
  if( (NULL != table) && (NULL != name) & (NULL != value) )
  {
    /* Add all as string values */
```

```
    result = GATTable_Add_String( table, (const void *) name, value );
  }

  /* Return to caller */
  return result;
}

static void printGATList_GATResource( GATList_GATResource resources )
{
  int counter;
  void **keys;
  float nextFloat;
  GATResult result;
  GATType nextType;
  char nextString[2048];
  GATTable_const table;
  GATResource resource;
  GATList_GATResource_Iterator end;
  GATList_GATResource_Iterator current;
  GATSoftwareResource softwareResource;
  GATSoftwareResourceDescription softwareResDes;

  /* Check parameters */
  if( NULL != resources )
  {
    /* Obtain Beginning Iterator */
    current = GATList_GATResource_Begin( resources );

    /* Check last call */
    if( NULL != current )
    {
      /* Obtain Ending Iterator */
      end = GATList_GATResource_End( resources );

      /* Check last call */
      if( NULL != end )
      {
        /* Loop over resources */
        while( (NULL != current) && (current != end) )
        {
          /* Obtain next GATResource */
          resource = GATList_GATResource_Get( current );

          /* Check last call */
          if( NULL != resource )
          {
              /* Convert the GATResource to a GATSoftwareResource
              softwareResource = GATResource_ToGATSoftwareResource( resource );
```

```
/* Obtain the GATSoftwareResourceDescription */
result =
 GATSoftwareResource_GetResourceDescription( softwareResource,
                                             &softwareResDes );

/* Check last call */
if( GAT_SUCCEEDED( result ) )
{
  /* Obtain GATTable */
  table =
    GATSoftwareResourceDescription_GetDescription( softwareResDes );

  /* Check last call */
  if( NULL != table )
  {
    /* Obtain keys */
    keys = GATTable_GetKeys( table );

    /* Check last call */
    if( NULL != keys )
    {
      /* Set counter */
      counter = 0;

      /* Loop over keys */
      while( NULL != keys[counter] )
      {
        /* Obtain value's type */
        nextType = GATTable_Get_ElementType( table, keys[counter]);

        /* Check type */
        if( GATfloat32 == nextType )
        {
          /* Obtain float */
          result =
           GATTable_Get_float( table, keys[counter], &nextFloat );

          /* Check last call */
          if( GAT_SUCCEEDED( result ) )
          {
            /* Print name=value */
            printf( "%s=%f\n", (char *) keys[counter], nextFloat );
          }
        }

        /* Check type */
        if( GATType_String == nextType )
        {
          /* Obtain string */
```

```
                        result =
                         GATTable_Get_String( table, keys[counter],
                                              nextString, 2048 );

                        /* Check last call */
                        if( GAT_SUCCEEDED( result ) )
                        {
                          /* Print name=value */
                          printf( "%s=%s\n", (char *) keys[counter], nextString );
                        }
                      }

                      /* Increament counter */
                      counter = counter + 1;
                    }

                    /* Print blank line to separate resources */
                    printf( "\n" );

                    /* Destroy keys */
                    GATTable_ReleaseKeys( table, keys );
                  }
                }

                /* Destroy GATSoftwareResourceDescription */
                GATSoftwareResourceDescription_Destroy( &softwareResDes );
              }
          }

          /* Obtain Next Iterator */
          current = GATList_GATResource_Next( current );
        }
      }
    }
  }
}
```

9 Interprocess Communication

9.1 Telephones of the Internet Age

So, you want to ring that long lost uncle twice removed on you father's side. What do you do? Pick up the phone and call him, talk, listen, and finally hang-up. The same is true in GAT. If one process wants to contact a second process, it goes through the exact same steps. Simple. GAT tries to make things as simple as possible, but no simpler. After all, what could be simpler than using one of these?

Figure 23: Typical phone circa 1936.

9.1.1 Calling

If you wanted to call that long lost uncle twice removed on you father's side, what you first have to do is get his phone number. One might try to divine this phone number from a weegee boad, channeling the spirit of your sixth cousin twice removed, the one who had only one outfit, the green pants and green jacket, but usually weegee boads aren't to effective. Trust me; I've tried. A more efficacious tack is to simply open up the phonebook and find the phone number. If he's listed, he's there. After finding the phone number, the next step is to connect to your uncle. This is easy; pick up the phone and dial those digits.

GAT makes interprocess communication just this simple. The calling process, when it wants to contact a second process, instead of looking in the phonebook for the proper number, looks in a `GATAdvertService` for a `GATEndpoint` instance. With such a `GATEndpoint`, the calling process can connect to the other process. All the calling process has to do is call the `GATEndpoint`'s "Connect" function.

9.1.2 Speaking and Listening

Speaking and listening to your long lost uncle twice removed on you father's side over the phone is easy. To speak just direct you voice to that little microphone, part of every phone these days, and lay forth. To listen just place your ear over the speaker and let nature take its course. With GAT, interprocess communication is also just that easy.

In calling a `GATEndpoint`'s "Connect" function the calling process is presented with an instance of the class `GATPipe`. The class `GATPipe` implements the interface `GATInterface_IStreamable`, an interface who's use we have already covered. Through this interface, the calling process can send or receive missives to its heart's content.

9.1.3 Hanging-Up

Finally, as all good things must come to an end, you have to hang up. For a phone one just pushes a single button[8]. For GAT one calls the "Close" function on `GATPipe`.

9.2 The Streaming Package

The GAT streaming package consists of three entities `GATEndpoint`, `GATPipe`, and `GATPipeListener`. The class `GATEndpoint` is used to create connections between two processes while the class `GATPipe` is used to allow the two so connected processes to exchange information. Finally, the interface `GATPipeListener` is used by the process which is being to connected to handle the resultant `GATPipe` instances.

9.2.1 Constructing and Destroying Endpoint Instances

As instances of the class `GATEndpoint` are used to create connections between processes, the first step that an application programmer has to take in using the streaming package is to learn how to create and destroy `GATEndpoint` instances. Fortunately, this is relatively simple. One only has to make a call to the following function

```
GATEndpoint GATEndpoint_Create(GATContext context, GATPreferences_const preferences)
```

to create a `GATEndpoint` instance. The arguments to this function, an instance of the class `GATContext` and the class `GATPreferences`, are classes we've encountered before. So, we will pass over them in silence. Upon success this function returns a `GATEndpoint` associated with

[8] As there are so many weird phones these days, I wouldn't bet my life on this last statement. For example, I know there exist wrist watch phones that transmit sound through the wearer's bones and are answered and hung-up by momentarily placing one's thumb and forefinger together.

the passed arguments. Upon failure it returns a NULL.

To destroy a so created GATEndpoint instance one calls the function

```
void GATEndpoint_Destroy(GATEndpoint *endpoint)
```

It takes a pointer to the GATEndpoint instance to destroy. Upon completion all resources held by the passed GATEndpoint instance are freed.

9.2.2 Listening on Endpoints

It's almost an afterthought with a phone, but to receive calls you have to listen for the phone to ring or whatever annoying bleeps you have the thing programmed to tourtue you, and those around you, with. But for GAT this is not an afterthought.

Before a process can be contacted by a second process using a GATEndpoint instance, it has to explicitly listen for such connections. This is done using one of two methods.

The first method used to listen for "incoming calls" is to invoke the function

```
GATResult GATEndpoint_Listen( GATEndpoint_const endpoint, GATPipe *peep)
```

This is a blocking function that waits until an "incoming call" has been received. It takes as its first argument a GATEndpoint instance created by the calling process. It's next argument is a pointer to a GATPipe, a class which we will cover below. It is through this pointer that this function returns a GATPipe instance which the process can use to converse with "incoming caller." Finally, this function returns a GATResult, covered in Appendix C, which indicates this function's completion status.

The second method used to listen for "incoming calls" is to invoke the function

```
GATResult GATEndpoint_AddGATPipeListener( GATEndpoint_const endpoint,
                                          GATPipeListener peepListener,
                                          void *listenerData)
```

This is a non-blocking function that does not wait until an "inccoming call" has been received. Again its first argument is a GATEndpoint instance created by the calling process. The second argument is a GATPipeListener and the final argument is simply a void * to untyped data.

A GATPipeListener is a function pointer with the following definition

```
typedef GATResult (*GATPipeListener)(void *, GATPipe);
```

It is used for call-backs. Whenever an "incoming call" is placed on the above endpoint, a call is made to the GATPipeListener passed to the above function. This GATPipeListener is passed a GATPipe instance along with the void * untyped data registered with the GATPipeListener. Finally, the function GATEndpoint_AddGATPipeListener returns a GATResult, a type covered in Appendix C, which indicates the function's completion status.

9.2.3 Advertising Endpoints

Once you've tricked the phone company in to comming over and installing your phone by taking an unpaid day off of work and sitting around the house all day in your underwear watching re-runs of "Days of Our Lives," you still can't get phone calls unless you tell someone your phone number. Usually, you just put your number in the phone book.

Lest we forget, Matthew 16:19 of the King James Bible: "... whatsoever thou shalt bind on earth shall be bound in heaven." So, whatsoever thou shalt bind with the phone company shall be bound with GAT. Once you've created a `GATEndpoint` instance no one can contact you unless you place the instance in a `GATAdvertService`. As the class `GATEndpoint` realizes the interface `GATInterface_IAdvertisable`, the process of placing a `GATEndpoint` instance is the exact same process already covered in Chapter 7. So, you know how to do this already.

9.2.4 Connecting on Endpoints

Once your friend has placed her phone number in the book, you can call them by simply finding the number in the phone book and letting your fingers do the walking. The same is true of GAT.

Once a process has placed a `GATEndpoint` in a `GATAdvertService`, anyone that wants to to "talk" to that process simply needs to find that `GATEndpoint` instance in the `GATAdvertService` then connect using the retrieved `GATEndpoint` instance.

As you already know how to retrieve instances from a `GATAdvertService`, this was covered in Chapter 7, we'll simply skip to the meat of the matter, connection. Upon retrieving a `GATEndpoint` instance from a `GATAdvertService`, one can connect to the process that placed the `GATEndpoint` there by calling the function

```
GATResult GATEndpoint_Connect( GATEndpoint_const endpoint, GATPipe *peep)
```

Its first argument is the `GATEndpoint` retrieved from the `GATAdvertService`. The second argument is a pointer to a `GATPipe`, a class which we will cover below. It is through this pointer that the function returns a `GATPipe` used to communicate with the remote process. Finally this function returns a `GATResult`, covered in Appendix C, which indicates this function's completion status.

9.2.5 Reading and Writing on Pipes

The class `GATPipe` realizes the interface `GATInterface_IStreamable`. As one will remember, in Chapter 5 we found that the class `GATFileStream` realized the interface `GATInterface_IStreamable`. As a result of this fact, we were able to read and write to `GATFileStream` instances using the various utility functions created expressly for that purpose. As `GATPipe` realizes the same interface, we can use these same utility functions to read and write to a `GATPipe` instance.

9.3 Some Useful Programs

9.3.1 A No So-Useful Echolalia Client and Server

As our first full-blown code example of the ground covered in this chapter, we will create an echolalia server and client. Echolalia is a psychiatric disorder in which causes those afflicted to mechanically repeat the uttering of other people, kinda of like a ditto-head, “Johnson, we should buy more shares of SCO.” “Yes boss, we should by more shares of SCO.”, but much more insidious.

Our echolalia server will simply repeat everything “said” by our echolalia client, and our echolalia client will “say” only those things passed to it via the command line. So, for example, we would first start the echolalia server as follows

```
% echolaliaserver
```

After the echolalia server is up and running we could start the echolalia client as follows

```
% echolaliaclient
```

The echolalia client will then wait for user input. For example, the user my type in the following, then hit return

```
Hey, you repeat everything I say.
```

The server would then print out

```
Hey, you repeat everything I say.
```

The idea is relatively simple, though it can be of use in tracking networking problems between a server and client. Let us now move on to code.

The full code for the echolalia client is as follows

```
#include <string.h>
#include <stdio.h>
#include "GAT.h"

static GATPipe Obtain_GATPipe( void );
static void Send_Messages( GATPipe pipe );

int main( int argc, char *argv[] )
{
  GATPipe pipe;

  /* Obtain  GATPipe */
  pipe = Obtain_GATPipe();

  /* Send Messages on GATPipe */
  if( NULL != pipe )
    Send_Messages( pipe );
```

```
  /* Return to OS */
  return 1;
}

static GATPipe Obtain_GATPipe( void )
{
  GATPipe pipe;
  GATResult result;
  const char *path;
  GATObject object;
  GATList_String paths;
  GATString stringPath;
  GATContext context;
  GATTable description;
  GATList_String_Iterator beginning;
  GATAdvertService advertService;

  /* Set GATPipe */
  pipe = NULL;

  /* Create GATContext */
  context = GATContext_Create();

  /* Check GATContext creation */
  if( NULL != context )
  {
    /* Create GATAdvertService */
    advertService = GATAdvertService_Create( context, NULL );

    /* Check GATAdvertService creation */
    if( NULL != advertService )
    {
      /* Create GATTable */
      description = GATTable_Create();

      /* Check GATTable creation */
      if( NULL != description )
      {
        /* Add Meta-Data to GATTable */
        result = GATTable_Add_String( description,
                                      (const void *) "name",
                                      "echolalia" );

        /* Check last call */
        if( GAT_SUCCEEDED( result ) )
        {
          /* Find GATPipe */
          result = GATAdvertService_Find( advertService, description, &paths );
```

```
    /* Check last call */
    if( GAT_SUCCEEDED( result ) )
    {
      /* Obtain Beginning Iterator */
      beginning = GATList_String_Begin( paths );

      /* Check last call */
      if( NULL != beginning )
      {
        /* Check for non-empty list */
        if( beginning != GATList_String_End( paths ) )
        {
          /* Get path */
          path = GATList_String_Get( beginning );

          /* Create GATString */
          stringPath = GATString_Create( path, strlen( path ) + 1, "ASCII" );

          /* Check GATString creation */
          if( NULL != stringPath )
          {
            /* Get Advertisable */
            result = GATAdvertService_GetAdvertisable( advertService,
                                                       stringPath,
                                                       &object );

            /* Check Last Call */
            if( GAT_SUCCEEDED( result ) )
            {
              /* Convert GATObject to GATPipe */
              pipe = GATObject_ToGATPipe( object );
            }

            /* Destroy GATString */
            GATString_Destroy( &stringPath );
          }
        }
      }

      /* Destroy GATList_String */
      GATList_String_Destroy( &paths );
    }
  }

  /* Destroy GATTable */
  GATTable_Destroy( &description );
}
```

```
      /* Destroy GATAdvertService */
      GATAdvertService_Destroy( &advertService );
    }

    /* Destroy GATContext */
    GATContext_Destroy( &context );
  }

  /* Return to caller */
  return pipe;
}

static void Send_Messages( GATPipe pipe )
{
  GATuint32 size;
  char input[2048];
  GATResult result;
  GATObject object;
  GATuint32 writtenBytes;

  /* Convert GATPipe to GATObject */
  object = GATPipe_ToGATObject( pipe );

  /* Loop Forever */
  while( GATTrue == GATTrue )
  {
    /* Read Input from User */
    if( NULL != fgets( input, sizeof( input ), stdin ) )
    {
      /* Obtain size */
      size = (GATuint32) (strlen( input ) + 1);

      /* Write to GATObject */
      result = GATStreamable_Write( object,
                                    (const void *) input,
                                    size,
                                    &writtenBytes );

      /* Check for Error */
      if( GAT_FAILED( result ) )
      {
        /* Print Error Message */
        printf( "An error occured in writing: %s\n", input );
      }
    }
    else
    {
      /* Print Error Message */
      printf( "An error occured in reading from stdin.\n" );
```

```
    }
  }

  /* Return to caller */
  return;
}
```

The full code for the echolalia server is as follows

```
#include <stdio.h>
#include <string.h>
#include "GAT.h"

static void Read_From_GATEndpoint( GATEndpoint endpoint );
static GATResult Advertise_GATEndpoint( GATContext context, GATEndpoint endpoint );

int main( int argc, char *argv[] )
{
  GATResult result;
  GATContext context;
  GATEndpoint endpoint;

  /* Create GATContext */
  context = GATContext_Create();

  /* Check GATContext creation */
  if( NULL != context )
  {
    /* Create GATEndpoint */
    endpoint = GATEndpoint_Create( context, NULL );

    /* Check GATEndpoint creation */
    if( NULL != endpoint )
    {
      /* Advertise GATEndpoint */
      result = Advertise_ GATEndpoint( context, endpoint );

      /* Check Last Call */
      if( GAT_SUCCEEDED( result ) )
      {
        /* Read GATEndpoint */
        Read_From_GATEndpoint( endpoint );
      }

      /* Destroy GATEndpoint */
      GATEndpoint_Destroy( &endpoint );
    }

    /* Destroy GATContext */
```

```
    GATContext_Destroy( &context );
  }

  /* Return to OS */
  return 1;
}

static GATResult Advertise_GATEndpoint( GATContext context, GATEndpoint endpoint )
{
  GATString path;
  GATResult result;
  GATObject object;
  GATTable description;
  GATAdvertService advertService;

  /* Set GATResult */
  result = GAT_MEMORYFAILURE;

  /* Create GATAdvertService */
  advertService = GATAdvertService_Create( context, NULL );

  /* Check GATAdvertService creation */
  if( NULL != advertService )
  {
    /* Convert GATEndpoint to GATObject */
    object = GATEndpoint_ToGATObject( endpoint );

    /* Create GATTable */
    description = GATTable_Create();

    /* Check GATTable create */
    if( NULL != description )
    {
      /* Create GATString */
      path = GATString_Create( "/usr/share/bin/echolaliad", 26, "ASCII" );

      /* Check GATString creation */
      if( NULL != path )
      {
        /* Add Meta-Data to GATTable */
        result =
         GATTable_Add_String( description, (const void *) "name", "echolalia" );

        /* Check last call */
        if( GAT_SUCCEEDED( result ) )
        {
          /* Add GATEndpoint to GATAdvertService */
          result =
           GATAdvertService_Add( advertService, object, description, path );
```

```
        }

        /* Destroy GATString */
        GATString_Destroy( &path );
      }

      /* Destroy GATTable */
      GATTable_Destroy( &description );
    }

    /* Destroy GATAdvertService */
    GATAdvertService_Destroy( &advertService );
  }

  /* Return to caller */
  return result;
}

static void Read_From_GATEndpoint( GATEndpoint endpoint )
{
  GATPipe pipe;
  GATResult result;
  char input[2048];
  GATObject object;
  GATuint32 readBytes;

  /* Wait For Incoming Call */
  result = GATEndpoint_Listen( endpoint, &pipe );

  /* Check last call */
  if( GAT_SUCCEEDED( result ) )
  {
    /* Convert GATPipe to GATObject */
    object = GATPipe_ToGATObject( pipe );

    /* Loop Until Failure */
    while( GAT_SUCCEEDED( result ) )
    {
      /* Read from GATObject */
      result = GATStreamable_Read( object, (void *) input, 2048, &readBytes );

      /* Check Last Call */
      if( GAT_SUCCEEDED( result ) )
      {
        /* Print Result */
        printf( "%s", input );
      }
    }
```

```
    /* Destroy GATPipe */
    GATPipe_Destroy( &pipe );
  }

  /* Return to caller */
  return;
}
```

10 Job Management

10.1 Job Management

As the culmination of your triple PhD's in Computer Science, Neuroscience, and Cognitive Science, you've cobbled together a wee-bit of code which you've taken to affectionally calling GAL 9000. It's a accurate simulation of a fully functioning human brain modeling all the brain's structure, soup to nuts, from neurotransmitter to encephalon. The only problem is, due to the recent round of funding cuts, all you have to run this code-art upon is your limp gimp VIC-20 which you ferreted out from the piles of rotting munchichis stashed in the attic of your childhood home. With only 3.5K of RAM available to run your code-art on, not even the likes of God could optimize memory usage enough to get you're "little me" up and running on the VIC. What is to be done?

Though you've not really been paying attention to all this "grid" hype as of late, one point has caught your attention - the grid's promise to allow the vast ever growing sea of the net, populated with hardware resources the likes of which you've only dreamt, to stand at your beckon command. Maybe this is where you should set your child free? The question is how?

The answer, GAT. GAT allows for an application programmer to start, stop, checkpoint, migrate, . . . computer processes across this endless sea of the net, and it allows the application programmer to do so, in typical GAT style, effortlessly. All of this functionality is squeezed tight within the job management package, who's study we now embark.

Quite generally, an application programmer need only describe, in a remarkably simple manner, the software they wish to run and the hardware on which they wish to run it, then the prestidigitator GAT handles the rest. So, Pygmalion you need not pray to Aphrodite to free you Galatea from the base stone of a VIC-20, GAT will do just fine.

10.2 The Job Management Package

The job management package consists of three main constituents: "resource descriptions," "jobs," and a "resource broker."

10.2.1 Resource Descriptions

As we found previously in Chapter 8, a resource description is, well, a description of a resource. There we introduced the resource description classes `GATResourceDescription`, `GATHardwareResourceDescription`, and `GATSoftwareResourceDescription`. To this trinity we add two new classes. The first is `GATSoftwareDescription`. This new entry is used to describe executables. So, for example, it would be used to describe your mania, Galatea. The second new class is `GATJobDescription`. This is used to describe a job and includes a description of the executable, a `GATSoftwareDescription` instance, along with a description of the hardware on which the job will run, a `GATResourceDescription` or `GATResource` instance.

10.2.2 Resource Broker

An old friend by now, the class `GATResourceBroker` was covered in Chapter 8, though there we didn't tell the whole story of this little gem. A `GATResourceBroker` can be used to do more than simply find and reserve resources. A `GATResourceBroker` can be used also to submit jobs. For example, it can be used to submit a job to a particular hardware resource, thus letting your Galatea break free of ties that bind, that frightful VIC-20.

10.2.3 Jobs

The last new class introduced as part of the job management package is, in all likelihood, the most important, the class `GATJob`. An instance of this class represents a physical job submitted to a resource management system. Submission, though, is not the end of the line for this little beauty; it slices, it dices, and it julians, whatever julianing may mean. We'll look at all various aspects of this `GATJob` class below.

10.2.4 Creating and Destroying SoftwareDescription Instances

The class `GATSoftwareDescription` is in many ways similar to a `GATSoftwareResourceDescription`; it is basically a container for a `GATTable` instance and this `GATTable` contains various name/value pairs which describe the executable the `GATSoftwareDescription` represents. The full set of supported name/value pairs is given in table 7

Name	**Type**	**Description**
`location`	`GATLocation`	Software location.
`arguments`	`List<String>`	Software command line arguments.
`environment`	`GATTable`	Software environment, names/values are C strings.
`stdin`	`GATFile`	Stdin from which the executable reads.
`stdout`	`GATFile`	Stdout to which the executable writes.
`stderr`	`GATFile`	Stderr to which the executable writes.
`pre-staged files`	`List<GATFile>`	Files staged to the resource before invocation.
`post-staged files`	`List<GATFile>`	Files staged from the resource after completion.

Table 7: Software Description: The minimum set of supported name/values.

As in the case of `GATSoftwareResourceDescription`, to construct an instance of the class `GATSoftwareDescription` one simply need a table. Explicitly one uses the function

```
GATSoftwareDescription GATSoftwareDescription_Create(GATTable_const attributes)
```

The first, and only, argument to this function is a `GATTable` instance which contains name/value pairs described in table 7. Upon success this function returns a `GATSoftwareDescription` instance corresponding to the passed `GATTable` instance. Upon failure it returns `NULL`.

To destroy a so created `GATSoftwareDescription` instance one uses the function

```
void GATSoftwareDescription_Destroy(GATSoftwareDescription *resource)
```

Upon completion this function releases an resources held by the passed `GATSoftwareDescription` instance.

10.2.5 Creating and Destroying JobDescription Instances

A `GATJobDescription`, unsurprisingly, describes a job. As in describing a job one usually needs to describe at least things: the executable which is to run and the hardware on which this executable is to run. As we saw previously, an executable is described by a `GATSoftwareDescription` instance and a hardware resource is described quite generally by a `GATHardwareResourceDescription` or more specifically by a `GATHardwareResource`. So, one might guess that a `GATJobDescription` is no more than an container for a `GATSoftwareDescription` and a `GATHardwareResourceDescription` or a `GATHardwareResource`, and if one guessed this, then they'd be right.

As a result, the construction of a `GATJobDescription` should be obvious. One can construct it using the function

```
GATJobDescription
GATJobDescription_Create_Description(GATContext context,
  GATSoftwareDescription_const software,
  GATResourceDescription_const resource)
```

which takes a `GATSoftwareDescription` describing the executable and a `GATResourceDescription` describing where to run the executable. Or on can use the function

```
GATJobDescription
GATJobDescription_Create(GATContext context,
  GATSoftwareDescription_const software,
  GATResource_const resource)
```

which takes a `GATSoftwareDescription` describing the executable and a `GATResource` indicating exactly where to run the executable. Both of these functions return a corresponding `GATJobDescription` on success and `NULL` on error.

There is only a single path to destruction, the function

```
void GATJobDescription_Destroy(GATJobDescription *resource)
```

which upon completion frees any resources held by the passed `GATJobDescription`.

10.2.6 Obtaining and Destroying Job Instances

An application programmer can not directly create a `GATJob` instance, she must first make oblation to the Ra of GAT. It is only through a call on an instance of the class `GATResourceBroker` that one can create a `GATJob` instance. In particular one must make a call to the function

```
GATResult
  GATResourceBroker_SubmitJob(GATResourceBroker broker,
    GATJobDescription_const description, GATJob *job)
```

The first argument is the `GATResourceBroker` used to create the `GATJob` instance. The next argument is a `GATJobDescription` which describes the desired `GATJob` . The final argument is a pointer to a `GATJob`. It is through this pointer that the function returns to the caller the desired `GATJob` instance. Finally, this function returns a `GATResult`, covered in Appendix C, indicating its completion status.

Destruction of such a `GATJob` instance is simple. Only one call to the function

```
void GATJob_Destroy(GATJob *resource)
```

and all resource held by the passed `GATJob` instance are released.

10.3 Obtaining the State of a Job Instance

A `GATJob` instance instance can be in various other states beyond simply submitted; what good would a job class be if all it did was stay in a submitted state. The various states a `GATJob` instance can be in are digramed in figure 24.

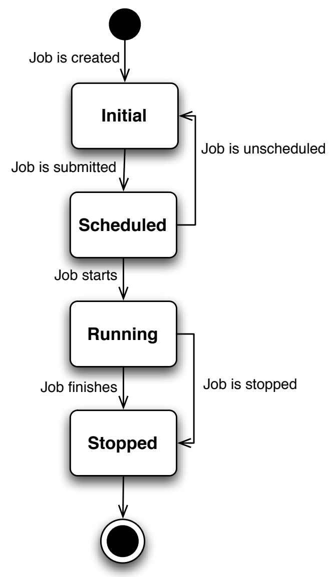

Figure 24: States of a GATJob instance.

After a `GATJob` instance has been created, it is in what is called the *initial state*. In this state the physical job corresponding to the `GATJob` instance has been submitted to an underlying resource management system, but has yet to be submitted to a specific hardware resource. A `GATJob` instance leaves the initial state when the corresponding physical job is submitted to a specific hardware resource. When the physical job is submitted to a specific hardware resource, the corresponding `GATJob` instance moves in to the *scheduled state*. A `GATJob` instance in the scheduled state can leave this state through one of two routes. First, it may leave this scheduled state by actually commencing to run, thus entering the *running state*. The other means by which is may leave the scheduled state is through an application unscheduling the `GATJob` instance, bringing it back in to the initial state. A `GATJob` instance in the running state can leave this state through one of two channels. First of all, the corresponding physical job may simply finish, leaving the `GATJob` instance in the *stopped state*. Second, an application may simply stop the `GATJob` instance leaving it again in the stopped state.

To actually determine what state a particular `GATJob` instance is in, one need only call the function

```
GATResult GATJob_GetState(GATJob_const object, GATJobState *state)
```

The first argument is the `GATJob` instance one wants to query for state information. The next argument is a pointer to a `GATJobState` through which the job's state is returned. The type `GATJobState` is an enumeration with the values and associated semantics enumerated in table 8.

GATJobState Value	Value's semantics
GATJobState_Unknown	The job's state can not be determined
GATJobState_Initial	Job is in the initial state
GATJobState_Scheduled	Job is in the scheduled state
GATJobState_Running	Job is in the running state
GATJobState_Stopped	Job is in the stopped state

Table 8: GATJobState enumeration values

Finally, our function of study, `GATJob_GetState`, returns a `GATResult`, covered in Appendix C, which indicates its completion status.

10.4 Obtaining the JobID of a Job Instance

A practice followed on all modern operating systems is that of assigning a unique identifier to each executing process. This is useful for any number of reasons. The most obvious of which is that it simply gives a means of uniquely referring to a particular process. Think how horrible it would be if you were without a name and had to live out the balance of your days answering only to "Hey, you over there. Yeah, you the funny looking one." Torture is other people.

To avoid all this mess GAT introduces a unique job ID for each `GATJob` instance. One obtains this job ID through a call to the function

```
GATResult GATJob_GetJobID(GATJob_const object, GATJobID_const *jobid)
```

This function's initial argument is the `GATJob` instance on wishes to query. The next argument is a pointer to a `GATJobID`. The class `GATJobID` is introduced solely to contain job ID's. It is through this pointer to a `GATJobID` that this function returns the job ID to the caller. Finally, this function returns a `GATResult`, covered in Appendix C, which indicates its completion status.

10.5 Obtaining the JobDescription of a Job Instance

After a one had used a `GATJobDescription` to create a `GATJob` instance one can obtain the `GATJobDescription` back from the `GATJob` instance. This is useful, for example, if one blind obtains a `GATJob` instance from a `GATAdvertService`. One obtains this `GATJobDescription` through a call to the function

```
GATResult GATJob_GetJobDescription(GATJob_const job, GATJobDescription_const *jd)
```

This function takes as its first argument a `GATJob` instance indicating the `GATJob` to query. The next argument is a pointer to a `GATJobDescription`. Through this pointer the function returns to the caller the apropos `GATJobDescription` instance. Finally, the function returns a `GATResult`, covered in Appendix C, indicating its completion status.

10.6 Obtaining Information about a Job Instance

Beyond the `GATJobDescription`. one can also obtain further information about a particular `GATJob` through a call to the function

```
GATResult GATJob_GetInfo(GATJob_const object, GATTable_const *jobinfo)
```

It's initial argument is the `GATJob` instance to query. The following argument is a pointer to a `GATTable` instance. Through this pointer this function returns to the caller a `GATTable` instance containing this further information, the details of which we will cover below. As one can see this function returns a `GATResult`, see Appendix C, which is used to indicate its completion status.

The `GATTable` instance obtained from the above function contains a set of various name/value pairs describing the job, adding a bit more detail to the info present in a `GATJobDescription`. The set of supported name/value pairs for such an info call are found in table 9

Name	**Type**	**Description**
`hostname`	`C String`	Name of the host on which the job is running.
`scheduletime`	`GATObject`	A GATTime indicating when the job was scheduled.
`starttime`	`GATObject`	A GATTime indicating when the job was started.
`stoptime`	`GATObject`	A GATTime indicating when the job was stopped.
`checkpointable`	`GATint16`	A 1 indicating the job is checkpointable, 0 otherwise.

Table 9: The supported set of name/value pairs for an info call.

In addition one should note that not all the various values in the table 9 make sense at all points in the life cycle of a `GATJob`. For example, if a `GATJob` instance is in an initial state, then the `starttime` name/value pair will not be present. Other obvious, similar restrictions hold on the various name/value pairs.

10.7 Un-Scheduling a Job Instance

For any number of reasons, one might decide after after a job is in the scheduled state to remove it from the scheduled state before it naturally reaches the running state. This would correspond to the state change labeled "Job is unscheduled" in the state diagram detail of figure 25.

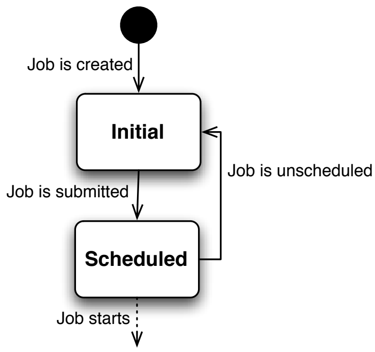

Figure 25: Detail of the GATJob state diagram.

As is apparent from figure 25, this "unscheduling state change" can only occur if a `GATJob` is in the scheduled state. Otherwise, such an "unscheduling state change" is not defined.

One can effect such an "unscheduling state change" on a `GATJob` through a call to the function

```
GATResult GATJob_UnSchedule(GATJob_const object)
```

The function's first and only argument is the `GATJob` instance on wishes to unschedule. As is standard, this function returns a GATResult, covered in Appendix C, which indicates its completion status. Upon successfully completing this function brings the passed `GATJob`, which must initially be in the scheduled state, in to the initial state. If the `GATJob` passed to this function is not in the scheduled state when passed, then this function will not complete successfully as the requested state change is undefined.

10.8 Stoping a Job Instance

GAT gives you the freedom to do whatever you see fit. For example, if you bring a job in to the running state, then decide, for whatever reason, be it logical or whimsy, that this job must be stopped by any means necessary, you can do it with GAT. One might realize, all to late, that one's job has some critical error, and running until the bloody end is all but pointless. So, one would like to preform a little euthanasia and put the job out to pasture. Such a state change would correspond to the state change labeled "Job is stopped" in the state diagram detail of figure 26

Apparent from figure 26, this "stopping state change" can only occur if a `GATJob` is in the running state. Otherwise, such a "stopping state change" is not defined.

To effect such a "stopping state change" one makes a call to the function

```
GATResult GATJob_Stop(GATJob object)
```

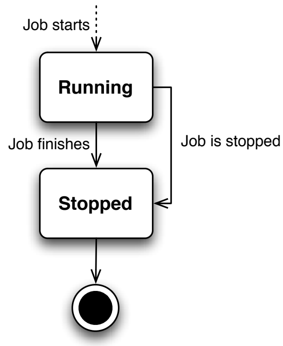

Figure 26: Detail of the GATJob state diagram.

The passed `GATJob` instance corresponds to the job one wishes to stop. Hence, this passed instance must be in a running state or this function will not complete successfully. This function returns a `GATResult`, covered in Appendix C, which indicates its completion status.

10.9 Checkpointing a Job Instance

As you've tired of writing endless reams of your own code, you've decide to outsource some of this work. When perusing some internet message groups you head of this set up with monkeys and type writers trying to reproduce the works of Shakespeare. What a great idea, you think. Why not do the same with code.

So, after many years of trekking through the deepest darkest of Africa to scout out the best of the best of the baboons, you're off to work. The only problem is the code these baboons produce, well, isn't of the highest quality. It keeps on crashing. Whoever wrote that post in that internet group was an idiot! But here you are with code that may be a bit unstable, but which you need to run. One way of lessening the burden in such a situation is to use checkpointing.

Checkpointing is a procedure in which a job's state is saved to long term storage. Hence, if a process is in the midst of a critical calculation, but may crash at any moment, one can save the state of this job by checkpointing it. After checkpointing a job, it can crash, but one will only lose any results calculated found after the job was checkpointed and before it crashed. So, you don't loose the whole run, just the little niggling bits around the edges.

GAT allows one to checkpoint properly instrumented jobs. (One can determine if a job is properly instrumented by calling the `GATJob_GetInfo` function and looking for the value of the key `checkpointable`.) However, a `GATJob` must be in the proper state before is can be checkpointed. As the `GATJob` must be running to save its state, one can see that it is only possible to checkpoint a `GATJob` in the running state. In particular, the *checkpointing state* is a substate of the running state as illustrated in figure 27.

One can checkpoint a properly instrumented job by making a call to the function

```
GATResult GATJob_Checkpoint(GATJob_const object)
```

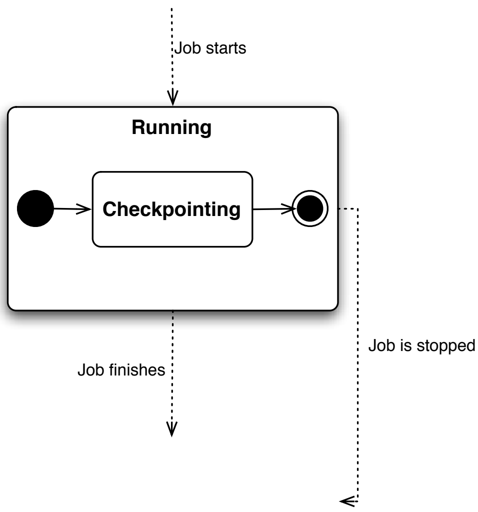

Figure 27: Detail of the GATJob state diagram.

The only argument to this function is a `GATJob` instance identifying the `GATJob` to checkpoint, and this function returns a `GATResult`, covered in Appendix C, which indicates its completion status. This is a non-blocking call; in other words, it does not wait until the checkpoint is actually completed. It simply delivers the checkpointing request to the job, then returns immediately. One should also note that this function will not complete successfully unless the passed `GATJob` is in a running state. As mentioned previously, the operation of checkpointing a `GATJob` is not even defined for a job not in the running state; so, it should come as no surprise that this function will fail when acting upon of non-running `GATJob`.

10.10 Migrating a Job Instance

Here's where things get interesting. Say you, Pygmalion, have somehow managed to shoehorn the encephalon of your beloved Galatea in to the 3.5K of RAM available in your salvaged VIC-20, but just as this Galatea is ever so slowly coming to life a new hardware resource comes on-line, the Earth Simulator. Through careful thought and much deliberation, you decide that it would likely be, well, a bit better to run your Galatea on the Earth Simulator instead of your once mighty VIC-20. However, you don't want to lose all the memories your Galatea has had living in the mortal coil of this VIC-20. Euthanasia is one thing, murder quite another. So, what you would really like to do is to take the current state of your Galatea save it, move it to the Earth Simulator, and start her there and stop the old job. Like going to sleep and waking up in the body of superman, if only. How can this be done? GAT!

GAT allows for this very thing, job migration, for properly instrumented jobs. (One can determine if a job is properly instrumented by calling the `GATJob_GetInfo` function and looking for the value of the key `checkpointable`. If the value is 1, then the job is properly instrumented.) One can migrate a job to a specified hardware resource through a call to the function

```
GATResult GATJob_Migrate(GATJob_const job,
```

```
                        GATHardwareResource_const hr,
                        GATJob *migratedJob)
```

This function will reconstitute the passed job using the state information saved in the previous call to the function `GATJob_Checkpoint` and stop the original job. It takes as its first argument a `GATJob` representing the job to migrate. Its second argument is a `GATHardwareResource` identifying the hardware resource to which the job should migrate. One can simply pass a `NULL` value for this second argument. This `NULL` signals that GAT should choose the new hardware resource. The final argument is a pointer to a `GATJob`. It is through this pointer that the function returns to the caller a `GATJob` instance corresponding to the migrated job. As is old hat by now, this function returns a `GATResult`, covered in Appendix C, which indicates its completion status.

10.11 Cloning a Job Instance

The `GATJob` class introduces a notion of cloning distinct from that found in all the various `GATObjects`. The class `GATJob` introduces this second notion of cloning to mean the exact same thing as migration, however, the initial job is not stopped. Thinking of your Galatea you can see this name is quite apropos. Upon cloning the running process that is Galatea you'll end up with two of her, the more the merrier.

Cloning a `GATJob` is accomplished through a call to the function

```
GATResult GATJob_CloneJob(GATJob_const object,
                          GATHardwareResource_const hr,
                          GATJob *cloned_job)
```

This function will reconstitute the passed job using the state information saved in the previous call to the function `GATJob_Checkpoint` but will not stop the original job. Its first argument is a `GATJob` identifying the job to clone. Its next argument is a `GATHardwareResource` specifying the hardware resource to which the job should migrate. One can simply pass a `NULL` value for this second argument. This `NULL` signals that GAT should choose the new hardware resource. The final argument is a pointer to a `GATJob`. It is through this pointer that the function returns to the caller a `GATJob` instance corresponding to the cloned job. This function returns a `GATResult`, covered in Appendix C, which indicates its completion status.

10.12 Some Useful Programs

10.12.1 See GAT run. Run GAT, run

In this chapter we will divert a bit from our usual track and present a rather large example program instead of several smaller programs. Though, we will keep with out habit of christening the example with a epithet suggestive of the program's functionality. Do you remember those children's books which you used to read in first grade to get your head around the complexities of this little miracle of the written word? Yes, I knew you'd remember. The one timeless poetic line from those days which has stuck with me since those days, "See Jane run. Run Jane, run." Beautific! In honour of this highest of poetics we christen this example of this chapter `gatrun`.

What does this `gatrun` thing do exactly you ask? Well, everything. Instead of giving a laundry list of the functionality of `gatrun`, we'll simply disperse with all these perfunctory ceremoniousness and just drop the manpage. Here it is

```
NAME
     gatrun - runs or otherwise manipulates a specified job

SYNOPSIS
     gatrun {-k jobid           |
             -s jobid           |
             -u jobid           |
             -c jobid           |
             vo file.hrd file.sd}

DESCRIPTION
     The command line utility gatrun is used to run or otherwise manipulate
     a specified job. In particular the main functionalities provided by the
     utility are as follows:

     - Killing a job with a specified jobid
     - Running a job specified through GATRL
     - Unscheduling a job with a particular jobid
     - Checkpointing a job with a specified jobid
     - Finding the status of a job with a particular jobid

     In addition this utility introduces a new specification, as if there
     were a dearth of them, called GATRL, which aims to be the most simple
     specification of a hardware resource description and a software
     description known to man.

     A GATRL file is simply a set of name/value pairs separated by an "="
     sign

     name=value

     Each name/value pair occupies a single line in a GATRL file. For example,
     to specify the name value pairs size=big and color=red and a GATRL file
     would contain the following lines

     size=big
     color=red

     The motto for GATRL is "GATRL it's not rocket science."

     Using the simple GATRL file format introduced above one can specify a
     hardware resource description and a software description. One simply
     uses the corresponding supported name/value pairs in the GATRL file.

     For example, if I wanted to specify a hardware resource description
```

```
using a GATRL file I might write something like this

memory.size=1024
machine.type=Power Macintosh
cpu.type=powerpc

Note that in a hardware resource description the value corresponding
to the name "memory.size" is a Float the utility gatrun takes care to
make sure that the supported names have values of the apropos type.
All other values are treated as strings.

One tricky point which arises is the conversion of a GATRL file in
to a software description. This is tricky as a software description
contains various classes. For example the name "location" has a value
of type GATLocation. For most of these types the mapping between a
GATRL file and the type is obvious. For example one know what is
implied by

location=http://www.google.com/index.html

or

stdin=file:///Users/leonardo/stdin.tmp

The tricky values to deal with correspond to the names

arguments - The value is of type GATList_String
environment - The value is of type GATTable

(Note, gatrun does not support the names "pre-staged files" and
"post-staged files".) Actually how the names "arguments" and
"environment" are dealt with in a GATRL file is also relatively
simple.

The name "arguments" has a value which is a ";" separated set of
strings. For example for the command ls one might pass

arguments=-l;/tmp/

Similarly, the value for the name "environment" is a ";" separated
set of name/value pairs where each name and value is separated by
an "=" sign. For example, to set the environment variable "HOME" to
"/Users/leonardo" and the environment variable "SHELL" to the value
"/bin/tcsh" one would have a line in the software description GATRL
file looking like

environment=HOME=/Users/leonardo;SHELL=/bin/tcsh

Quite simple really.
```

```
     The following options are available:

     -k jobid
            Kills the job specified by the passed jobid

     -s jobid
            Prints out the status of the job specified by the passed jobid

     -u jobid
            Unschedules the job specified by the passed jobid

     -c jobid
            Checkpoints the job specified by the passed jobid

EXAMPLES
     The following shows how to kill the job with jobid 132

         gatrun -k 132

     To find the status of the job with jobid 423 one would use

         gatrun -s 423

     To unschedule the job with jobid 498 one would use

         gatrun -u 498

     To checkpoint the job with jobid 9AAT67 one would use

         gatrun -c 9AAT67

     Starting the job specified by the GATRL hardware resource description
     file File.hrd and the GATRL software description file File.sd within
     the virtual organization gridlab.org would look as follows

         gatrun gridlab.org File.hrd File.sd

DIAGNOSTICS
     The gatrun utility exits 0 on success, and >0 if an error occurs.

COMPATIBILITY
     The gatrun utility is compatible with itself, maybe.

SEE ALSO
     gatsaunter(1), gatamble(1), gatperambulation(1), gatstroll(1)

STANDARDS
     The gatrun utility conforms to IEEE Std 1003.1-2001 (``POSIX.1''), not!
```

```
HISTORY
     An gatrun command appeared in Version 1 of this manual

BUGS
     To maintain backward compatibility with the wheel we have rounded the
     jutting corners of this sharp utility to protect the bunglesome users.
```

Ok time to jump in with both feet. Here's the code

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#include "GAT.h"

static void GATRun_PrintUsage( void );
static GATResult GATRun_AdvertiseJob( GATJob job );
static GATResult GATRun_Run( int argc, char *argv[] );
static GATResult GATRun_Run_Run( int argc, char *argv[] );
static GATResult GATRun_Run_Kill( int argc, char *argv[] );
static GATResult GATRun_Run_Status( int argc, char *argv[] );
static GATResult GATRun_Run_Unschedule( int argc, char *argv[] );
static GATResult GATRun_Run_Checkpoint( int argc, char *argv[] );
static GATResult GATRun_CommandLineValid( int argc, char *argv[] );
static GATResult GATRun_GetJobByID( const char *jobid, GATJob *job );
static GATResult GATRun_ParseGATRL( const char *file, GATTable *table );
static GATResult GATRun_CreateJob( int argc, char *argv[], GATJob *job );
static GATResult GATRun_CommandLineValidCaseOne( int argc, char *argv[] );
static GATResult GATRun_CommandLineValidCaseTwo( int argc, char *argv[] );
static  GATResult GATRun_GATTable_AddFloat( GATTable table, const char *name, const char *va
static  GATResult GATRun_GATTable_AddGATFile( GATTable table, const char *name, const char *
static  GATResult GATRun_GATTable_AddGATTable( GATTable table, const char *name, const char
static  GATResult GATRun_GATTable_AddGATLocation( GATTable table, const char *name, const ch
static  GATResult GATRun_GATTable_AddGATList_String( GATTable table, const char *name, const

int main( int argc, char *argv[] )
{
  int returnValue;
  GATResult result;

  /* Check Command Line Arguments */
  result = GATRun_CommandLineValid( argc, argv );
  if( GAT_FAILED( result ) )
  {
    /* Print Usage */
    GATRun_PrintUsage();
  }
  else
```

```
  {
    /* Run gatrun */
    result = GATRun_Run( argc, argv );
  }


  /* Set returnValue */
  if( GAT_SUCCEEDED( result ) )
  {
    returnValue = 0;
  }
  else
  {
    returnValue = 1;
  }

  /* Return to OS */
  return returnValue;
}

/**
 * Checks the command line arguments are valid.
 *
 * @param argc Number of command line arguments
 * @param argv Command line arguments
 * @return GATResult indicating completion status
 */
static GATResult GATRun_CommandLineValid( int argc, char *argv[] )
{
  GATResult result;

  /* Assume Invalid Parameter */
  result = GAT_INVALID_PARAMETER;

  /* Check Number of Arguments: Case One -- Kill, Status, Unschedule, and Checkpoint */
  if( 3 == argc )
  {
    /* Check Kill, Status, Unschedule, and Checkpoint Options */
    result = GATRun_CommandLineValidCaseOne( argc, argv );
  }

  /* Check Number of Arguments: Case Two -- Run */
  if( 5 == argc )
  {
    /* Check Run Options */
    result = GATRun_CommandLineValidCaseTwo( argc, argv );
  }

  /* Return to caller */
```

```
  return result;
}

/**
 * Checks the command line arguments are valid for the Kill, Status, Unschedule,
 * and Checkpoint command line arguments.
 *
 * @param argc Number of command line arguments
 * @param argv Command line arguments
 * @return GATResult indicating completion status
 */
static GATResult GATRun_CommandLineValidCaseOne( int argc, char *argv[] )
{
  GATResult result;

  /* Assume Invalid Parameter */
  result = GAT_INVALID_PARAMETER;

  /* Check Kill */
  if( 0 == strcmp( "-k", argv[1] ) )
    result = GAT_SUCCESS;

  /* Check Status */
  if( 0 == strcmp( "-s", argv[1] ) )
    result = GAT_SUCCESS;

  /* Check Unschedule */
  if( 0 == strcmp( "-u", argv[1] ) )
    result = GAT_SUCCESS;

  /* Check Checkpoint */
  if( 0 == strcmp( "-c", argv[1] ) )
    result = GAT_SUCCESS;

  /* Return to caller */
  return result;
}

/**
 * Checks the command line arguments are valid for the Run command
 * line arguments.
 *
 * @param argc Number of command line arguments
 * @param argv Command line arguments
 * @return GATResult indicating completion status
 */
static GATResult GATRun_CommandLineValidCaseTwo( int argc, char *argv[] )
{
  /*
```

```
   * Note:
   *
   * We assume all specified files and vo's are valid; thus the existence of the
   * correct number of command line arguments is sufficient to stamp the args
   * as valid.
   *
   */

  /* Return to caller */
  return GAT_SUCCESS;
}

/**
 * Prints the usage for gatrun
 */
static void GATRun_PrintUsage( void )
{
  printf("NAME\n");
  printf("     gatrun - runs or otherwise manipulates a specified job \n");
  printf("\n");
  printf("SYNOPSIS\n");
  printf("     gatrun {-k jobid           | \n");
  printf("             -s jobid           | \n");
  printf("             -u jobid           | \n");
  printf("             -c jobid           | \n");
  printf("             vo file.hrd file.sd}\n");
  printf("\n");
  printf("DESCRIPTION\n");
  printf("     The command line utility gatrun is used to run or otherwise manipulate\n");
  printf("     a specified job. In particular the main functionalities provided by the\n");
  printf("     utility are as follows:\n");
  printf("\n");
  printf("     - Killing a job with a specified jobid  \n");
  printf("     - Running a job specified through GATRL \n");
  printf("     - Unscheduling a job with a particular jobid\n");
  printf("     - Checkpointing a job with a specified jobid\n");
  printf("     - Finding the status of a job with a particular jobid\n");
  printf("\n");
  printf("     In addition this utility introduces a new specification, as if there\n");
  printf("     were a dearth of them, called GATRL, which aims to be the most simple\n");
  printf("     specification of a hardware resource description and a software \n");
  printf("     description known to man. \n");
  printf("\n");
  printf("     A GATRL file is simply a set of name/value pairs separated by an \"=\"\n");
  printf("     sign\n");
  printf("\n");
  printf("     name=value\n");
  printf("\n");
  printf("     Each name/value pair occupies a single line in a GATRL file. For example, \n"
```

```
printf("     to specify the name value pairs size=big and color=red and a GATRL file\n");
printf("     would contain the following lines\n");
printf("\n");
printf("     size=big\n");
printf("     color=red\n");
printf("\n");
printf("     The motto for GATRL is \"GATRL it's not rocket science.\"\n");
printf("\n");
printf("     Using the simple GATRL file format introduced above one can specify a\n");
printf("     hardware resource description and a software description. One simply \n");
printf("     uses the corresponding supported name/value pairs in the GATRL file.\n");
printf("\n");
printf("     For example, if I wanted to specify a hardware resource description\n");
printf("     using a GATRL file I might write something like this\n");
printf("\n");
printf("     memory.size=1024\n");
printf("     machine.type=Power Macintosh\n");
printf("     cpu.type=powerpc\n");
printf("     \n");
printf("     Note that in a hardware resource description the value corresponding\n");
printf("     to the name \"memory.size\" is a Float the utility gatrun takes care to\n");
printf("     make sure that the supported names have values of the apropos type.\n");
printf("     All other values are treated as strings. \n");
printf("\n");
printf("     One tricky point which arises is the conversion of a GATRL file in\n");
printf("     to a software description. This is tricky as a software description\n");
printf("     contains various classes. For example the name \"location\" has a value\n");
printf("     of type GATLocation. For most of these types the mapping between a\n");
printf("     GATRL file and the type is obvious. For example one know what is \n");
printf("     implied by\n");
printf("\n");
printf("     location=http://www.google.com/index.html\n");
printf("\n");
printf("     or \n");
printf("\n");
printf("     stdin=file:///Users/leonardo/stdin.tmp\n");
printf("\n");
printf("     The tricky values to deal with correspond to the names \n");
printf("\n");
printf("     arguments - The value is of type GATList_String\n");
printf("     environment - The value is of type GATTable\n");
printf("\n");
printf("     (Note, gatrun does not support the names \"pre-staged files\" and \n");
printf("     \"post-staged files\".) Actually how the names \"arguments\" and\n");
printf("     \"environment\" are dealt with in a GATRL file is also relatively \n");
printf("     simple.\n");
printf("\n");
printf("     The name \"arguments\" has a value which is a \";\" separated set of\n");
printf("     strings. For example for the command ls one might pass \n");
```

```
printf("\n");
printf("     arguments=-l;/tmp/\n");
printf("\n");
printf("     Similarly, the value for the name \"environment\" is a \";\" separated\n");
printf("     set of name/value pairs where each name and value is separated by\n");
printf("     an \"=\" sign. For example, to set the environment variable \"HOME\" to\n");
printf("     \"/Users/leonardo\" and the environment variable \"SHELL\" to the value\n");
printf("     \"/bin/tcsh\" one would have a line in the software description GATRL\n");
printf("     file looking like\n");
printf("\n");
printf("     environment=HOME=/Users/leonardo;SHELL=/bin/tcsh\n");
printf("\n");
printf("     Quite simple really. \n");
printf("\n");
printf("     The following options are available:\n");
printf("\n");
printf("     -k jobid \n");
printf("            Kills the job specified by the passed jobid\n");
printf("\n");
printf("     -s jobid \n");
printf("            Prints out the status of the job specified by the passed jobid \n");
printf("\n");
printf("     -u jobid \n");
printf("            Unschedules the job specified by the passed jobid \n");
printf("\n");
printf("     -c jobid \n");
printf("            Checkpoints the job specified by the passed jobid \n");
printf("\n");
printf("EXAMPLES\n");
printf("     The following shows how to kill the job with jobid 132\n");
printf("         \n");
printf("         gatrun -k 132\n");
printf("\n");
printf("     To find the status of the job with jobid 423 one would use\n");
printf("\n");
printf("         gatrun -s 423\n");
printf("\n");
printf("     To unschedule the job with jobid 498 one would use\n");
printf("\n");
printf("         gatrun -u 498\n");
printf("\n");
printf("     To checkpoint the job with jobid 9AAT67 one would use\n");
printf("\n");
printf("         gatrun -c 9AAT67\n");
printf("\n");
printf("     Starting the job specified by the GATRL hardware resource description\n");
printf("     file File.hrd and the GATRL software description file File.sd within \n");
printf("     the virtual organization gridlab.org would look as follows\n");
printf("\n");
```

```
  printf("         gatrun gridlab.org File.hrd File.sd\n");
  printf("\n");
  printf("DIAGNOSTICS\n");
  printf("     The gatrun utility exits 0 on success, and >0 if an error occurs.\n");
  printf("\n");
  printf("COMPATIBILITY\n");
  printf("     The gatrun utility is compatible with itself, maybe. \n");
  printf("\n");
  printf("SEE ALSO\n");
  printf("     gatsaunter(1), gatamble(1), gatperambulation(1), gatstroll(1) \n");
  printf("\n");
  printf("STANDARDS\n");
  printf("     The gatrun utility conforms to IEEE Std 1003.1-2001 (``POSIX.1''), not!\n");
  printf("\n");
  printf("HISTORY\n");
  printf("     An gatrun command appeared in Version 1 of this manual\n");
  printf("\n");
  printf("BUGS\n");
  printf("     To maintain backward compatibility with the wheel we have rounded the\n");
  printf("     jutting corners of this sharp utility  to protect the bunglesome users.\n");

  /* Return to caller */
  return;
}

/**
 * The entry point for this utility after the format of the command line arguments
 * has been checked
 *
 * @param argc Number of command line arguments
 * @param argv Command line arguments
 * @return GATResult indicating completion status
 */
static GATResult GATRun_Run( int argc, char *argv[] )
{
  GATResult result;

  /* Assume Invalid Parameter */
  result = GAT_INVALID_PARAMETER;

  /* Branch on Number of Arguments: Case One -- Kill, Status, Unschedule, and Checkpoint */
  if( 3 == argc )
  {
    /* Run Kill */
    if( 0 == strcmp( "-k", argv[1] ) )
    result = GATRun_Run_Kill( argc, argv );

    /* Run Status */
    if( 0 == strcmp( "-s", argv[1] ) )
```

```
    result = GATRun_Run_Status( argc, argv );

    /* Run Unschedule */
    if( 0 == strcmp( "-u", argv[1] ) )
    result = GATRun_Run_Unschedule( argc, argv );

    /* Run Checkpoint */
    if( 0 == strcmp( "-c", argv[1] ) )
     result = GATRun_Run_Checkpoint( argc, argv );
  }

  /* Branch on Number of Arguments: Case Two -- Run */
  if( 5 == argc )
  {
    /* Run Run */
    result = GATRun_Run_Run( argc, argv );
  }

  /* Return to caller */
  return result;
}

/**
 * Kills the job with the jobid argv[2]
 *
 * @param argc Number of command line arguments
 * @param argv Command line arguments
 * @return GATResult indicating completion status
 */
static GATResult GATRun_Run_Kill( int argc, char *argv[] )
{
  GATJob job;
  GATResult result;

  /* Get GATJob by ID */
  result = GATRun_GetJobByID( argv[2], &job );

  /* Check last call */
  if( GAT_SUCCEEDED( result ) )
  {
    /* Stop GATJob */
    result = GATJob_Stop( job );

    /* Destroy GATJob */
    GATJob_Destroy( &job );
  }

  /* Check result */
  if( GAT_SUCCEEDED( result ) )
```

```
  {
    /* Print status message */
    print( "Killed job %s\n", argv[2] );
  }
  else
  {
    /* Print status message */
    print( "Could not kill job %s\n", argv[2] );
  }

  /* Return to caller */
  return result;
}

/**
 * Prints the status of the job with the jobid argv[2]
 *
 * @param argc Number of command line arguments
 * @param argv Command line arguments
 * @return GATResult indicating completion status
 */
static GATResult GATRun_Run_Status( int argc, char *argv[] )
{
  GATJob job;
  GATResult result;
  GATJobState state;

  /* Get GATJob by ID */
 result = GATRun_GetJobByID( argv[2], &job );

  /* Check last call */
  if( GAT_SUCCEEDED( result ) )
  {
    /* Get State of GATJob */
    result = GATJob_GetState( job,  &state );

    /* Check last call */
    if( GAT_SUCCEEDED( result ) )
    {
      /* Print Unknown State */
      if( GATJobState_Unknown == state )
        printf( "The job %s is in an unknown state.\n", argv[2] );

      /* Print Initial State */
      if( GATJobState_Initial == state )
        printf( "The job %s is in the initial state.\n", argv[2] );

      /* Print Scheduled State */
      if( GATJobState_Scheduled == state )
```

```
        printf( "The job %s is in the scheduled state.\n", argv[2] );

      /* Print Running State */
      if( GATJobState_Running == state )
        printf( "The job %s is in the running state.\n", argv[2] );

      /* Print Stopped State */
      if( GATJobState_Stopped == state )
        printf( "The job %s is in the stopped state.\n", argv[2] );
    }

    /* Destroy GATJob */
    GATJob_Destroy( &job );
  }

  /* Check result */
  if( GAT_FAILED( result ) )
  {
    /* Print status message */
    print( "Could not determine status of job %s\n", argv[2] );
  }

  /* Return to caller */
  return result;
}

/**
 * Unschedules the job with the jobid argv[2]
 *
 * @param argc Number of command line arguments
 * @param argv Command line arguments
 * @return GATResult indicating completion status
 */
static GATResult GATRun_Run_Unschedule( int argc, char *argv[] )
{
  GATJob job;
  GATResult result;

  /* Get GATJob by ID */
  result = GATRun_GetJobByID( argv[2], &job );

  /* Check last call */
  if( GAT_SUCCEEDED( result ) )
  {
    /* Unschedule GATJob */
    result = GATJob_UnSchedule( job );

    /* Destroy GATJob */
    GATJob_Destroy( &job );
```

```
  }

  /* Check result */
  if( GAT_SUCCEEDED( result ) )
  {
    /* Print status message */
    print( "Unscheduled job %s\n", argv[2] );
  }
  else
  {
    /* Print status message */
    print( "Could not unschedule job %s\n", argv[2] );
  }

  /* Return to caller */
  return result;
}

/**
 * Checkpoints the job with the jobid argv[2]
 *
 * @param argc Number of command line arguments
 * @param argv Command line arguments
 * @return GATResult indicating completion status
 */
static GATResult GATRun_Run_Checkpoint( int argc, char *argv[] )
{
  GATJob job;
  GATResult result;

  /* Get GATJob by ID */
  result = GATRun_GetJobByID( argv[2], &job );

  /* Check last call */
  if( GAT_SUCCEEDED( result ) )
  {
    /* Checkpoint GATJob */
    result = GATJob_Checkpoint( job );

    /* Destroy GATJob */
    GATJob_Destroy( &job );
  }

  /* Check result */
  if( GAT_SUCCEEDED( result ) )
  {
    /* Print status message */
    print( "Checkpointed job %s\n", argv[2] );
  }
```

```
  else
  {
    /* Print status message */
    print( "Could not checkpoint job %s\n", argv[2] );
  }

  /* Return to caller */
  return result;
}

/**
 * Obtains the job with the specified jobid from the GATAdvertService
 *
 * @param jobid The jobid
 * @param job The obtained GATJob
 * @return GATResult indicating completion status
 */
static GATResult GATRun_GetJobByID( const char *jobid, GATJob *job )
{
  GATString path;
  GATResult result;
  char const *cpath;
  GATObject object;
  GATTable metadata;
  GATList_String paths;
  GATContext context;
  GATList_String_Iterator beginning;
  GATAdvertService advertService;

  /* Assume Invalid Parameter */
  result = GAT_INVALID_PARAMETER;

  /* Check Parameter */
  if( NULL !=  job )
  {
    /* Assume Memory Failure */
    result = GAT_MEMORYFAILURE;

    /* Create GATContext */
    context = GATContext_Create();

    /* Check creation */
    if( NULL != context )
    {
      /* Create GATAdvertService */
      advertService = GATAdvertService_Create( context, NULL );

      /* Check creation */
      if( NULL != advertService )
```

```
{
  /* Create GATTable */
  metadata = GATTable_Create();

  /* Check creation */
  if( NULL != metadata )
  {
    /* Add jobid=jobid to metadata */
    result = GATTable_Add_String( metadata, (const void *) "jobid", jobid );

    /* Check last call */
    if( GAT_SUCCEEDED( result ) )
    {
      /* Find GATJobs */
      result = GATAdvertService_Find( advertService, metadata, &paths );

      /* Check last call */
      if( GAT_SUCCEEDED( result ) )
      {
        /* Obtain GATList_String_Iterator */
        beginning = GATList_String_Begin( paths );

        /* Check last call */
        if( NULL != beginning )
        {
          /* Check Existence of GATJob */
          if( beginning != GATList_String_Begin(paths) )
          {
            /* Obtain path as C string */
            cpath = GATList_String_Get( beginning );

            /* Check last call */
            if( NULL != cpath )
            {
              /* Create GATString path */
              path = GATString_Create( cpath, strlen( cpath ) + 1, "ASCII" );

              /* Check GATString creation */
              if( NULL != path )
              {
                /* Get Advertisable */
                result = GATAdvertService_GetAdvertisable( advertService, path, &objec

                /* Check last call */
                if( GAT_SUCCEEDED( result ) )
                {
                  /* Convert GATObject to GATJob */
                  *job = GATObject_ToGATJob( object );
                }
```

```
                /* Destroy GATString */
                GATString_Destroy( &path );
              }
              else
              {
                /* Indicate Memory Failure */
                result = GAT_MEMORYFAILURE;
              }
            }
            else
            {
              /* Indicate Memory Failure */
              result = GAT_MEMORYFAILURE;
            }
          }
          else
          {
            /* Indicate No Such GATJob Exists */
            result = GAT_NO_MATCHING_RESOURCE;
          }
        }
        else
        {
          /* Indicate Memory Failure */
          result = GAT_MEMORYFAILURE;
        }

        /* Destroy GATList_String */
        GATList_String_Destroy( &paths );
      }
    }

    /* Destroy GATTable */
    GATTable_Destroy( &metadata );
  }

  /* Destroy GATAdvertService */
  GATAdvertService_Destroy( &advertService );
}

/* Destroy GATContext */
GATContext_Destroy( &context );
}
}

/* Return to caller */
return result;
}
```

```
/**
 * Runs the job within the virtual organization argv[1] on hardware
 * described by the hardware resource description resulting from
 * parsing the GATRL file argv[2]. The job itself is described by
 * the software description resulting from parsing the GATRL
 * file argv[3]. The resultant GATJob is placed in the GATAdvert
 * Service with the meta-data
 *
 * jobid = <GATJob's JobID>
 *
 * so as to allow other processes to access the job.
 *
 * @param argc Number of command line arguments
 * @param argv Command line arguments
 * @return GATResult indicating completion status
 */
static GATResult GATRun_Run_Run( int argc, char *argv[] )
{
  GATJob job;
  GATResult result;

  /* Create GATJob */
  result = GATRun_CreateJob( argc, argv, &job );

  /* Check GATJob creation */
  if( GAT_SUCCEEDED( result ) )
  {
    /* Print Status */
    printf( "Successfully submitted the job.\n" );

    /* Advertise GATJob */
    result = GATRun_AdvertiseJob( job );

    /* Check Last Call */
    if( GAT_SUCCEEDED( result ) )
    {
      /* Print status */
      printf( "Successfully stored the job so other processes can access it.\n" );
    }

    /* Destroy GATJob */
    GATJob_Destroy( &job );
  }

  /* Return to caller */
  return result;
}
```

```
/**
 * Runs the job within the virtual organization argv[1] on hardware
 * described by the hardware resource description resulting from
 * parsing the GATRL file argv[2]. The job itself is described by
 * the software description resulting from parsing the GATRL
 * file argv[3] and returns the GATJob through job.
 *
 * @param argc Number of command line arguments
 * @param argv Command line arguments
 * @param job The GATJob corresponding to the passed arguments
 * @return GATResult indicating completion status
 */
static GATResult GATRun_CreateJob( int argc, char *argv[], GATJob *job )
{
  GATString vo;
  GATResult result;
  GATContext context;
  GATJobDescription jobDescription;
  GATTable softwareDescriptionTable;
  GATResourceBroker resourceBroker;
  GATSoftwareDescription softwareDescription;
  GATTable hardwareResourceDescriptionTable;
  GATResourceDescription resourceDescription;
  GATHardwareResourceDescription hardwareResourceDescription;

  /* Assume Invalid Parameter */
  result = GAT_INVALID_PARAMETER;

  /* Check Parameter */
  if( NULL !=  job )
  {
    /* Assume Memory Failure */
    result = GAT_MEMORYFAILURE;

    /* Create GATContext */
    context = GATContext_Create();

    /* Check GATContext Creation */
    if( NULL != context )
    {
      /* Create GATString */
      vo = GATString_Create( argv[1], strlen( argv[1] ) + 1, "ASCII" );

      /* Check GATString Creation */
      if( NULL != vo )
      {
        /* Create GATResourceBroker *
        resourceBroker = GATResourceBroker_Create( context, NULL, vo );
```

```
/* Check GATResourceBroker Creation */
if( NULL != resourceBroker )
{
  /* Obtain GATTable for GATHardwareResourceDescription */
  result = GATRun_ParseGATRL( argv[2], &hardwareResourceDescriptionTable );

  /* Check Last Call */
  if( GET_SUCCEEDED( result ) )
  {
    /* Assume Memory Failure */
    result = GAT_MEMORYFAILURE;

    /* Create GATHardwareResourceDescription */
    hardwareResourceDescription = GATHardwareResourceDescription_Create( hardwareRes

    /* Check GATHardwareResourceDescription Creation */
    if( NULL != hardwareResourceDescription )
    {
      /* Obtain GATTable for GATSoftwareDescription */
      result = GATRun_ParseGATRL( argv[3], &softwareDescriptionTable );

      /* Check Last Call */
      if( GAT_SUCCEEDED( result ) )
      {
        /* Assume Memory Failure */
        result = GAT_MEMORYFAILURE;

        /* Create GATSoftwareDescription */
        softwareDescription = GATSoftwareDescription_Create( softwareDescriptionTabl

        /* Check GATSoftwareDescription Creation */
        if( NULL != softwareDescription )
        {
          /* Convert GATHardwareResourceDescription to GATResourceDescription */
          resourceDescription = GATHardwareResourceDescription_ToGATResourceDescript

          /* Create GATJobDescription */
          jobDescription = GATJobDescription_Create( context, softwareDescription, r

          /* Check Last Call */
          if( NULL != jobDescription )
          {
            /* Create GATJob */
            result = GATResourceBroker_SubmitJob( resourceBroker, jobDescription, jo

            /* Destroy GATJobDescription */
            GATJobDescription_Destroy( &jobDescription );
          }
```

```
                /* Destroy GATSoftwareDescription */
                GATSoftwareDescription_Destroy( &softwareDescription );
              }
            }

            /* Destroy GATHardwareResourceDescription */
            GATHardwareResourceDescription_Destroy( &hardwareResourceDescription );
          }

          /* Destroy GATTable */
          GATTable_Destroy( &hardwareResourceDescriptionTable );
        }

        /* Destroy GATResourceBroker */
        GATResourceBroker_Destroy( &resourceBroker );
      }

      /* Destroy GATString */
      GATString_Destroy( &vo );
    }

    /* Destroy GATContext */
    GATContext_Destroy( &context );
  }
}

  /* Return to caller */
  return result;
}

/**
 * Parses the specified GATRL file into a set of name/value pairs and places
 * those name value pairs in the passed GATTable
 *
 * @param fileName The specified GATRL file
 * @param table The GATTable into which names/values are parsed.
 * @return GATResult indicating completion status
 */
static GATResult GATRun_ParseGATRL( const char *fileName, GATTable *table )
{
  FILE *file;
  char *value;
  char *name;
  GATResult result;
  char nextLine[2048];

  /* Assume Invalid Parameter */
  result = GAT_INVALID_PARAMETER;
```

```
/* Check Parameter */
if( NULL !=  table )
{
  /* Create GATTable */
  *table = GATTable_Create();

  /* Check GATTable Creation */
  if( NULL = (*table) )
  {
    /* Assume IO Error */
    result = GAT_IO_ERROR;

    /* Open file */
    file = fopen( fileName, "r" );

    /* Check Last Call */
    if( NULL == file )
    {
      /* Assume Invalid GATRL */
      result = GAT_UNKNOWN_FORMAT;

      /* Read Next Line */
      while( NULL != fgets( nextLine, 2048, file ) )
      {
        /* Read name */
        name = strtok( nextLine, "=" );

        /* Check Last Call */
        if( NULL != name )
        {
          /* Read value */
          value = strtok( NULL, "=" );

          /* Check Last Call */
          if( NULL != value )
          {
            /* Add name/values to table */
            if( 0 == strcmp( "memory.size", name ) )
            {
              /* Add memory.size, value is a float */
              result = GATRun_GATTable_AddFloat( *table, name, value );
            } else if( 0 == strcmp( "memory.accesstime", name ) )
            {
              /* Add memory.accesstime, value is a float */
              result = GATRun_GATTable_AddFloat( *table, name, value );
            } else if( 0 == strcmp( "memory.str", name ) )
            {
              /* Add memory.str, value is a float */
              result = GATRun_GATTable_AddFloat( *table, name, value );
```

```
} else if( 0 == strcmp( "cpu.speed", name ) )
{
  /* Add cpu.speed, value is a float */
  result = GATRun_GATTable_AddFloat( *table, name, value );
} else if( 0 == strcmp( "disk.size", name ) )
{
  /* Add disk.size, value is a float */
  result = GATRun_GATTable_AddFloat( *table, name, value );
} else if( 0 == strcmp( "disk.accesstime", name ) )
{
  /* Add disk.accesstime, value is a float */
  result = GATRun_GATTable_AddFloat( *table, name, value );
} else if( 0 == strcmp( "disk.str", name ) )
{
  /* Add disk.str, value is a float */
  result = GATRun_GATTable_AddFloat( *table, name, value );
} else if( 0 == strcmp( "location", name ) )
{
  /* Add location, value is a GATLocation */
  result = GATRun_GATTable_AddGATLocation( *table, name, value );
} else if( 0 == strcmp( "arguments", name ) )
{
  /* Add arguments, value is a GATList_String */
  result = GATRun_GATTable_AddGATList_String( *table, name, value );
} else if( 0 == strcmp( "environment", name ) )
{
  /* Add environment, value is a GATTable */
  result = GATRun_GATTable_AddGATTable( *table, name, value );
} else if( 0 == strcmp( "stdin", name ) )
{
  /* Add stdin, value is a GATFile */
  result = GATRun_GATTable_AddGATFile( *table, name, value );
} else if( 0 == strcmp( "stdout", name ) )
{
  /* Add stdout, value is a GATFile */
  result = GATRun_GATTable_AddGATFile( *table, name, value );
} else if( 0 == strcmp( "stderr", name ) )
{
  /* Add stderr, value is a GATFile */
  result = GATRun_GATTable_AddGATFile( *table, name, value );
} else
{
  /* Add name, value is a C String */
  result = GATTable_Add_String( *table, (const void *) name, value);
}

/* On Failure, break while Loop*/
if( GAT_FAILED( result ) )
{
```

```
                break;
              }
            }
          }
        }

        /* Close file */
        fclose( file );
      }
    }
  }

  /* Return to Caller */
  return result;
}

/**
 * Adds the name/value pair to the passed table interpreting the
 * value as a C String representation of a GATfloat32.
 *
 * @param table The GATTable which to augment
 * @param name The name of the name/value pair
 * @param value The value of the name/value pair
 * @return A GATResult indicating completion status
 */
static  GATResult GATRun_GATTable_AddFloat( GATTable table, const char *name, const char *va
{
  GATfloat32 floatValue;

  /Convert value to float */
  floatValue = (GATfloat32) atof( value );

  /* Add to table */
  return GATTable_Add_float( table, (const void *) name, floatValue );
}

/**
 * Adds the name/value pair to the passed table interpreting the
 * value as a C String representation of a GATLocation.
 *
 * @param table The GATTable which to augment
 * @param name The name of the name/value pair
 * @param value The value of the name/value pair
 * @return A GATResult indicating completion status
 */
static  GATResult GATRun_GATTable_AddGATLocation( GATTable table, const char *name, const ch
{
  GATResult result;
  GATObject object;
```

```
  GATLocation location;

  /* Set result to a memory failure */
  result = GAT_MEMORYFAILURE;

  /* Create GATLocation */
  location = GATLocation_Create( value );

  /* Check GATLocation Creation */
  if( NULL != location )
  {
    /* Convert GATLocation to GATObject */
    object = GATLocation_ToGATObject( location );

    /* Add object to table */
    result = GATTable_Add_GATObject( table, (const void *) name, object );

    /* Destroy GATLocation */
    GATLocation_Destroy( &location );
  }

  /* Return to Caller */
  return result;
}

/**
 * Adds the name/value pair to the passed table interpreting the
 * value as a C String representation of a GATList_String, elements
 * in value are semi-colon delimited.
 *
 * @param table The GATTable which to augment
 * @param name The name of the name/value pair
 * @param value The value of the name/value pair
 * @return A GATResult indicating completion status
 */
static  GATResult GATRun_GATTable_AddGATList_String( GATTable table, const char *name, const
{
  char *nextString;
  GATResult result;
  GATObject object;
  GATList_String strings;
  GATList_String_Iterator beginning;

  /* Set result to a memory failure */
  result = GAT_MEMORYFAILURE;

  /* Create GATList_String */
  strings = GATList_String_Create();
```

```
  /* Check GATList_String Creation */
  if( NULL != strings )
  {
    /* Obtain GATList_String_Iterator */
    beginning = GATList_String_Begin( strings );

    /* Check Last Call */
    if( NULL != beginning )
    {
      /* Set result to a unknown format */
      result = GAT_UNKNOWN_FORMAT;

      /* Obtain First nextString */
      nextString = strtok( value, ";" );

      /* Tokenize value */
      while( NULL != nextString )
      {
        /* Insert nextString to strings */
        result = GATList_String_Insert( strings, beginning, nextString );

        /* On Failure, break while loop */
        if( GAT_FAILED( result ) )
          break;

        /* Obtain next nextString */
        nextString = strtok( NULL, ";" );
      }

      /* Check result Call */
      if( GAT_SUCCEEDED( result ) )
      {
        /* Convert GATList_String to GATObject */
        object = GATList_String_ToGATObject( strings );

        /* Add object to table */
        result = GATTable_Add_GATObject( table, (const void *) name, object );
      }
    }

    /* Destroy GATList_String */
    GATList_String_Destroy( &strings );
  }

  /* Return to Caller */
  return result;
}

/**
```

```
 * Adds the name/value pair to the passed table interpreting the
 * value as a C String representation of a GATTable, name/value
 * pairs in value are semi-colon delimited and the name and
 * value are separated by an equals.
 *
 * @param tableOne The GATTable which to augment
 * @param name The name of the name/value pair
 * @param value The value of the name/value pair
 * @return A GATResult indicating completion status
 */
static  GATResult GATRun_GATTable_AddGATTable( GATTable tableOne, const char *name, const ch
{
  GATResult result;
  char *nextValue;
  char *nextName;
  GATObject object;
  GATTable tableTwo;

  /* Set result to a memory failure */
  result = GAT_MEMORYFAILURE;

  /* Create GATTable */
  tableTwo = GATTable_Create();

  /* Check GATTable Creation */
  if( NULL != tableTwo )
  {
    /* Set result to a unknown format */
    result = GAT_UNKNOWN_FORMAT;

    /* Obtain First nextName */
    nextName = strtok( value, ";=" );

    /* Tokenize Value */
    while( NULL != nextName )
    {
      /* Obtain Next nextValue */
      nextValue =  strtok( NULL, ";=" );

      /* Check Last Call */
      if( NULL == nextValue )
      {
        /* Set result to a unknown format */
        result = GAT_UNKNOWN_FORMAT;

        /* Break while */
        break;
      }
```

```
      /* Add nextName and nextValue to tableTwo */
      result = GATTable_Add_String( tableTwo, (const void *) nextName, nextValue );

      /* Check Last Call */
      if( GAT_FAILED( result ) )
      {
        /* Break while */
        break;
      }

      /* Obtain Next nextName */
      nextName = strtok( NULL, ";=" );
    }

    /* Check result Call */
    if( GAT_SUCCEEDED( result ) )
    {
      /* Convert GATTable to GATObject */
      object = GATTable_ToGATObject( tableTwo );

      /* Add object to tableOne */
      result = GATTable_Add_GATObject( tableOne, (const void *) name, object );
    }

    /* Destroy GATTable */
    GATTable_Destroy( &tableTwo );
  }

  /* Return To Caller */
  return result;
}

/**
 * Adds the name/value pair to the passed table interpreting the
 * value as a C String representation of a GATFile.
 *
 * @param table The GATTable which to augment
 * @param name The name of the name/value pair
 * @param value The value of the name/value pair
 * @return A GATResult indicating completion status
 */
static  GATResult GATRun_GATTable_AddGATFile( GATTable table, const char *name, const char *
{
  GATFile file;
  GATResult result;
  GATObject object;
  GATContext context;

  /* Set result to a memory failure */
```

```
  result = GAT_MEMORYFAILURE;

  /* Create GATContext */
  context = GATContext_Create();

  /* Check GATContext Creation */
  if( NULL != context )
  {
    /* Create GATFile */
    file = GATFile_Create_Name( context, value, NULL );

    /* Check GATFile Creation */
    if( NULL != file )
    {
      /* Convert GATFile to GATObject */
      object = GATFile_ToGATObject( file );

      /* Add object to table */
      result = GATTable_Add_GATObject( table, (const void *) name, object );

      /* Destroy GATFile */
      GATFile_Destroy( &file );
    }

    /* Destroy GATContext */
    GATContext_Destroy( &context );
  }

  /* Return to Caller */
  return result;
}

/**
 * Advertises the passed GATJob in the GATAdvertService with the meta-data
 *
 * jobid=<jobid>
 *
 * and the POSIX path
 *
 * /tmp/jobs/<jobid>
 *
 * @param job The GATJob to advertise
 * @return A GATResult indicating completion status
 */
static GATResult GATRun_AdvertiseJob( GATJob job )
{
  GATString path;
  GATResult result;
  GATObject object;
```

```
char tmpPath[2048];
GATTable metadata;
GATContext context;
GATJobID_const jobid;
const char *jobidString;
GATAdvertService advertService;

/* Set result to a memory failure */
result = GAT_MEMORYFAILURE;

/* Create GATContext */
context = GATContext_Create();

/* Check GATContext Creation */
if( NULL != context )
{
  /* Create GATAdvertService */
  advertService = GATAdvertService_Create( context, NULL );

  /* Check GATAdvertService Creation */
  if( NULL != advertService )
  {
    /* Get GATJobID */
    result = GATJob_GetJobID( job, &jobid );

    /* Check Last Call */
    if( GAT_SUCCEEDED( result ) )
    {
      /* Add prefix to tmpPath */
      strcpy( tmpPath, "/tmp/jobs/" );

      /* Obtain jobidString */
      jobidString = GATString_GetBuffer( jobid ); /* May not be ASCII !!! */

      /* Add suffix to tmpPath */
      strcat( tmpPath, jobidString );

      /* Set result to a memory failure */
      result = GAT_MEMORYFAILURE;

      /* Create GATString */
      path = GATString_Create( tmpPath, strlen( tmpPath) + 1, "ASCII" );

      /* Check GATString Creation */
      if( NULL != path )
      {
        /* Create GATTable */
        metadata = GATTable_Create();
```

```
          /* Check GATTable Creation */
          if( NULL != metadata )
          {
            /* Add jobid=<jobid> to metadata */
            result = GATTable_Add_String( metadata, (const void *) "jobid", jobidString );

            /* Check Last Call */
            if( GAT_SUCCEEDED( result ) )
            {
              /* Convert GATJob to GATObject */
              object = GATJob_ToGATObject( job );

              /* Add object to advertService at path with metadata */
              result = GATAdvertService_Add( advertService, object, metadata, path );
            }

            /* Destroy GATTable */
            GATTable_Destroy( &metadata );
          }

          /* Destroy GATString */
          GATString_Destroy( &path );
        }
      }

      /* Destroy GATAdvertService */
      GATAdvertService_Destroy( &advertService );
    }

    /* Destroy GATContext */
    GATContext_Destroy( &context );
  }

  /* Return to caller */
  return result;
}
```

11 Monitoring

11.1 Spying: A User's Guide

As everyone knows, in the infamous words of Steve Jobs,

Good artists copy, great artists steal.

The makers of GAT took this lesson too heart when creating the monitoring package. This package was created to allow the application programmer to spy on the sleeping, unsuspecting masses, allowing her to peek, unhindered by the burden of distance, in to the workings of GAT class instances everywhere. All the application programmer has to do is ask, and GAT provides the dirt on alll the instances swimming in the grid ocean.

So, upon whom did GAT model this proclivity for the peeled eye you ask. None other than the best of the best, the spy's spy, and that is, of course, Lancelot Link Secret Chimp and not that half-wit 007.

Figure 28: The one, the only Lancelot Link Secret Chimp.

In pouring over the back reels of Lancelot, the makers of GAT found that the one true Agent held to this singular spying path: Lance would first inveigle one of his minions in to planting a sensor in a critical region of the desired target; he'd determine which sensors his lackey had successfully planted, and Lance would monitor these targets using the embedded sensors. The GAT team would like to thank Lance for showing us *the way* as GAT uses these same three steps – learned from the one true master, Lance – to monitor the variegated instances of GAT classes.

Various players, just beyond the application programmer's purview, plant sensors in critical regions of desired GAT targets. For example, one of these peons might place a sensor on a hardware resource which measures the amount of free memory, or another flunky might place a sensor which reports on free disk space.

Once these sensors are placed, the application programmer can quiz GAT to detect which sensors are planted at a particular location. For example, the application programmer can query a GAT instance to determine which of the various sensors have been installed. The application programmer might get a list back saying that the RAM sensor is there, but the free disk space sensor didn't get planted on this particular target.

Finally, the application programmer can use the sensors embedded in critical regions of the desired target to monitor the in's and out's of this target's daily doings. For example, the application programmer, after obtaining notice that the RAM sensor had been successfully embedded, could then go about using this sensor to remotely monitor the amount of RAM free on the target of interest. Lance, we thank you.

11.2 The Monitoring Package

The monitoring package splays its tentacles throughout the entirety of GAT. Instances of manifold GAT classes are capable of being monitored. In point of fact, any class which realizes the interface `GATInterface_IMonitorable` is capable of being monitored, and there are so very many such instances. All such classes depicted in figure 29 implement this interface.

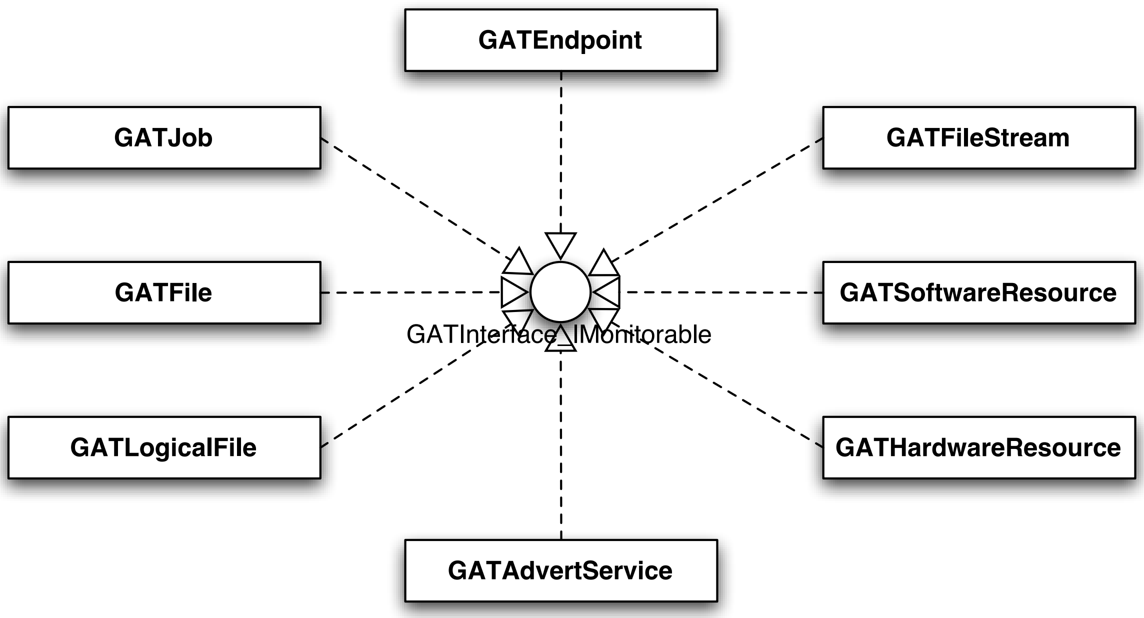

Figure 29: Classes realizing the interface GATInterface_IMonitorable.

Beyond containing various classes which can be monitored, the monitoring package contains assorted classes which facilitate the act of monitoring.

In particular, the monitoring package contains the class `GATMetric`. As one will recall, the second step Lance would take, when faced with the prospect of spying on some arch enemies such as the dreaded Dr. Strangemind, is to determine which sensors his lackeys successfully planted. In the parallel universe of GAT, one would query a monitorable instance to determine which sensors have been planted in this monitorable. Upon doing so, one is presented with a list of `GATMetric` instances. Each of these `GATMetric` instances corresponds to a sensor placed in the monitorbale of interest.

The monitoring package also contains another abstraction which goes by the curious name `GATMetricListener`. (Actually, the name isn't all that bizarre; rather, it's quite apropos. Nepotism – I actually decided on this name.) This is the means by which an application may listen to a monitorable. In our Lance scenario above, it might correspond to a digital readout indicating the temperate in Dr. Strangemind's lair. In the alternative reality of GAT, a `GATMetricListener` is a function pointer type. Functions which are of this type, i.e. functions with the correct signature, can be registered with a monitorable and upon being so registered can listen for a sensor sending out signal of the monitoable's day-to-day dealings.

11.2.1 Obtaining and Destroying Metric Instances

An application programmer can not directly create instances of the class `GATMetric`. They must rather ask a monitorable instance to create such `GATMetric` instances for them through a "GetMetrics" call. Explicitly, the call

```
GATResult GATMonitorable_GetMetrics(GATObject_const object,
                                    struct GATList_GATMetric *metrics)
```

would be used to obtain a list of `GATMetric` instances corresponding to all the various sensors associated with the `GATObject` instance.

In gory detail, this function takes as its first argument an instance of the class `GATObject`, which, for the correct operation of this function, must be a monitorable. This instance represents the `GATObject` that one wishes to examine for planted sensors. The next argument is a pointer to a `GATList_GATMetric`. It it through this pointer that the function returns a list of `GATMetric` instances each of which corresponds to a sensor supported by the passed `GATObject` instance. Finally, this function returns a `GATResult`, a type covered in Appendix C, which indicates this function's completion status.

To take this new function once around the block we can consider a code snippet which could be used to obtain all the `GATMetric` instances corresponding to the sensors planted in a `GATFile` instance.

```
GATFile file;
GATResult result;
GATObject object;
GATList_GATMetric metrics;

file = ...
object = GATFile_ToGATObject( file );
```

```
result = GATMonitorable_GetMetrics( object, &metrics );

if( GAT_SUCCEEDED( result ) )
{
  /* Do something with the metrics */
}
```

Simple, no?

To destroy `GATMetric` instances is also easy as 1-2-3. One uses the function

```
void GATMetric_Destroy(GATMetric *metric)
```

This function takes as its first argument a pointer to the `GATMetric` instance to be put-out to pasture. Upon completion all the resources held by this `GATMetric` instance are released. This function should be called upon any `GATMetric` instance one wishes to dispose of.

In addition, what is often of more use in the case of a `GATMetric` instance, is to use the function

```
void GATList_GATMetric_Destroy( GATList_GATMetric *metrics )
```

which destroys the `GATList_GATMetric` pointed to by the passed pointer. In addition, it destroys the contents of the pointed to `GATList_GATMetric`, i.e. it will call `GATMetric_Destroy` on all the contained `GATMetric` instances.

11.2.2 Examining a Metric

Upon obtaining a `GATMetric` instance one naturally wants poke and probe at it a bit to see just what has been hauled up from the deep. This interrogation usually is accomplished through the use of a set of helper functions, which we will examine here, that are built expressly for this purpose. However, before going in to the details of these helper functions lets take a step back and examine conceptually our pull.

A `GATMetric` returned from a call to the function `GATMonitorable_GetMetrics` indicates that a particular sensor has been placed in the monitorable of interest and this sensor is ready to be used for monitoring this monitorable. In the GAT sphere such a sensor is identified through a set of various data. The most conspicuous member of this data set is the name of the sensor. Each `GATMetric` instance returned from a call to the function `GATMonitorable_GetMetrics` contains the name of the sensor to which it associated. Beyond the name, this most visible piece of information, a `GATMetric` instance contains a set of various name/value pairs with serve to further identify the sensor to which the `GATMetric` instance corresponds. These various name/value pairs can be accessed through a `GATTable` instance, a class covered in Appendix F.

So, one can see that a `GATMetric` requires various utility functions which allow one to probe the `GATMetric` innards. In particular, one needs a function to obtain the name of a `GATMetric`, a function to obtain the value type corresponding to a particular name/value pair, and a function to obtain the value of a particular name/value pair. In addition, GAT throws in, for the price of free, as in beer, an extra function which allows the application programmer to obtain a `GATTable` which contains all the various name/value pairs describing a particular sensor. Let examine in detail these functions.

Obtaining a Metric's Name

To obtain the name of a `GATMetric` instance one uses the function

```
const char *GATMetric_GetName(GATMetric_const metric)
```

This function takes as its first argument the `GATMetric` instance which to query and returns a pointer to a `const char` containing a C string representation of the passed `GATMetric`'s name.

Obtaining a Metric's Parameter Type

A `GATMetric` instance contains various name/value pairs which identify the sensor to which the metric corresponds. The names in these name/value pairs are always C strings. However, the values may be any type enumerated by the enumeration `GATType`, covered in Appendix A. To determine the type of the value in a particular name/value pair on calls the function

```
GATType
  GATMetric_GetParameterTypeByName(GATMetric_const metric,
    char const *name)
```

The first argument to this function is the `GATMetric` to examine. The second argument is the name, expressed as a C string, of the parameter one wishes to examine the type of. This function returns a `GATType` indicating the type of the value corresponding to the passed name.

Obtaining a Metric's Parameter Value

Now that we know how to obtain the type of a given value we can actually move on to obtaining this value. This is accomplished through the use of the function

```
GATResult
  GATMetric_GetParameterByName(GATMetric_const metric, char const *name,
    GATType type, void *buffer, GATuint32 size)
```

The first parameter passed to this function is the `GATMetric` to query. The next argument is the name of the parameter, expressed as a C string, the value of which we want to obtain. The argument subsequent is a `GATType`, covered in Appendix A, which indicates the type of the value that we are seeking. This would, for example, be obtained through a call to the function `GATMetric_GetParameterTypeByName`. Next this function is passed a `void *` which points to the memory used to contain the result of this call, the value we are seeking. The final argument is a `GATuint32` which indicates the number of bytes pointed to by the previous argument. As is usual, this function returns a `GATResult`, covered in Appendix C, indicating this function's completion status.

To clear up the use of this function well take a look at how it is used in concert with the previous function we examined. A code snippet which uses both functions together is as follows

```
GATType type;
const char *name;
GATMetric metric;
GATResult result;
GATuint32 value;

name = ...
metric = ...

type = GATMetric_GetParameterTypeByName( metric, name );

if( GATType_GATuint32 == type )
{
  result = GATMetric_GetParameterByName( metric, name, type, (void *) value, 32 );

  if( GAT_SUCCEEDED( result ) )
  {
    /* Do something with value */
  }
}
```

Obtaining a Metric's Names and Values

Instead of nickel-and-dimeing yourself to death obtaining name/value pairs contained in a `GATMetric` instance one by one GAT, the spy master's toolkit, allows you to obtain all of them in one go. This is accomplished through the use of this function

```
GATResult
  GATMetric_GetParameters(GATMetric_const metric, GATTable *table)
```

It takes as its first argument the `GATMetric` to examine. Its second argument is a pointer to a `GATTable`, a class covered in Appendix F. It is through this pointer that the function returns to the caller a `GATTable` instance containing all of the various name/value pairs contained within this `GATMetric` instance. Per-usual this function returns a `GATResult`, covered in Appendix **??**, that indicates its completion status.

Obtaining a Metric's Unit

When a sensor, corresponding to a `GATMetric`, actually reports a measurement, it reports this measurement using a particular measuring unit. For example, if it were to measure a length, it might report the result in cm. If it were to measure a time, it might report the result in seconds. If it were to measure an area, it might report its result in barns[9]. A `GATMetric` can be queried for the unit in which its corresponding sensor makes measurements through use of the function

```
char const *
  GATMetric_GetUnit(GATMetric_const metric)
```

[9]A barn is $10^{-28} m^2$.

which takes as its first and only argument the `GATMetric` to query and returns a C string representation of the unit the corresponding sensor measures in.

Obtaining a Metric's Value Type

Beyond obtaining the unit the sensor measures in, one can also obtain a `GATType` corresponding to the value returned by the sensor through use of the function

```
GATType
  GATMetric_GetValueType(GATMetric_const metric)
```

This function takes as its first argument the `GATMetric` to query for information and it returns a `GATType` indicating the `GATType` corresponding to the value returned by the `GATMetric`.

11.2.3 Adding MetricListeners

As we know now the cabalistic arcanum of obtaining and examining `GATMetric` instances, we'll now bring ourselves to the next rung on the ladder to GAT spiritual enlightenment and learn the Om of registering to receive information from an planted sensor.

To register to receive information from a planted sensor we need to specify a slew of information. We need to specify the monitorable we want to spy on, where this information – when collected should go, and which sensor to use for this monitoring. In addition to this required information, we need also specify a few technical bits of data which we will clarify below. The function used for all this magic is

```
GATResult
  GATMonitorable_AddMetricListener(GATObject object,
    GATMetricListener listener, void *listener_data, GATMetric metric,
    GATuint32 *cookie)
```

It's first argument is a `GATObject`. This `GATObject` instance is the instance we are registering to receive information from and it must realize the interface `GATInterface_IMonitorable`. The next argument to this function is a `GATMetricListener`. The `GATMetricListener` is a function pointer which specifies the function to receive information from the planted sensor. In particular, this type is defined as follows

```
typedef GATResult (*GATMetricListener)(void *, GATMetricEvent);
```

The next argument to the function `GATMonitorable_AddMetricListener` is a `void *` this points to "callback data" that the `GATMetricListener` may require and is passed as the first argument to the passed `GATMetricListener` when it is called with sensor information. The function's next argument is a `GATMetric` which specifies the sensor to be monitored. The final argument is a pointer to a `GATuint32`. Through this pointer a `GATuint32` is returned to the caller. As the registered `GATMetricListener` is a function pointer and not a full-blown GAT class, it does not have an "Equals" method. Hence, when it comes time to de-register this `GATMetricListener` one has no means of identifying this particular combination of `GATMetricListener`, sensor, listener data, . . . So, to solve this problem GAT introduces this "cookie" which uniquely identifies this particular combination of `GATMetricListener`, sensor, listener data, . . . This "cookie" is used to identify this combination when de-registering this listener. As is usual, this function returns a `GATResult`, covered in Appendix C, which indicates it completion status.

Before we examine the art of removing a `GATMetricListener` instance we'll take a little break and focus a bit on exactly what a `GATMetricListener` and `GATMetricEvent` are. In passing we mentioned that a `GATMetricListener` is a function pointer defined as follows

```
typedef GATResult (*GATMetricListener)(void *, GATMetricEvent);
```

A `GATMetricListener` when registered will be called at apropos times with sensor information and `void *` data. The sensor information is encapsulated in the `GATMetricEvent` instanced passed to this function and the `void *` passed to this function is a pointer to the "listener data" originally passed when this function was registered. Lets examine this `GATMetricEvent` class in more detail.
Beyond the standard member functions of a GAT class, a `GATMetricEvent` has a function which allows for one to obtain the source of the `GATMetricEvent`

```
GATObject_const
  GATMetricEvent_GetSource(GATMetricEvent_const metric_event)
```

This function takes a `GATMetricEvent`, the instance one wishes to find the source of, and returns a `GATObject`, the source of the passed `GATMetricEvent`. For example, this might be a `GATHardwareResource` which is being monitored or a `GATFile` instance which is being monitored. The class `GATMetricEvent` also has a function

```
GATType
  GATMetricEvent_GetValueType(GATMetricEvent_const metric_event)
```

which allows one to find the type of value measured by the sensor; for example, a sensor may return a `int` indicating the number of iterations or a `char *` indicating the name of the last Russian czar. This function takes a `GATMetricEvent`, the instance one wishes to examine, and returns a `GATType`, covered in Appendix A, that indicates the type of the value measured by the corresponding sensor. To actually obtain this value one uses the function

```
GATResult
  GATMetricEvent_GetValue(GATMetricEvent_const metric_event, void *buffer,
    GATuint32 size)
```

This function is similar to `GATMetric_GetParameterByName`. It takes as its first argument the `GATMetricEvent` whose value we wish to peek at. The next argument is a `void *` pointing to a region in memory which is to contain the value upon successful completion of this function, and the final argument if the size in bytes of the region pointer to by the previous argument. As always, this function returns a `GATResult`, covered in Appendix C, which indicates the completion status of this function. Beyond obtaining the value the sensor measured one can obtain the time at which this measurement was made using the function

```
GATTime_const
  GATMetricEvent_GetEventTime(GATMetricEvent metric_event)
```

which takes the `GATMetricEvent` instance to obtain the measurement time of and returns a `GATTime` instance indicating the time at which the corresponding sensor's measurement was made. In addition, one can obtain the `GATMetric` corresponding to this `GATMetricEvent`, i.e. a `GATMetric` equivalent to the `GATMetric` registered with `GATMetricListener` which is being called with this `GATMetricEvent`. All of this is done through the function

```
GATMetric_const
  GATMetricEvent_GetMetric(GATMetricEvent_const metric_event)
```

which takes the `GATMetricEvent` whose `GATMetric` we are trying to obtain and returns a corresponding `GATMetric`.

11.2.4 Removing MetricListeners

Now we have the process of adding `GATMetricListener` down cold removing a `GATMetricListeners` is actually simple. We use the function

```
GATResult
  GATMonitorable_RemoveRegisteredMetric(GATObject object,
    GATMetric metric, GATuint32 cookie)
```

The first argument is a `GATObject` which is the monitorable to remove the `GATMetricListener` from, and hence must realize the interface `GATInterface_IMonitorable`. The following argument is a `GATMetric` indicating the `GATMetric` with which the `GATMetricListener` to remove was registered. The final argument is a `GATuint32` indicating the cookie with which the `GATMetricListener` was registered. The function returns a `GATResult`, covered in Appendix C, which indicates this function's completion status.

11.2.5 Actually Listening

As we've now explored how to register and de-register a `GATMetricListener`, one may ask, especially for a single threaded application, "When is the registered `GATMetricListener` actually called?" For in a single threaded application the application would have to explicitly hand over the thread of control to code which would call the registered `GATMetricListener`'s. The application hands over the thread of control to code which calls the registered `GATMetricListener`'s through the following call

```
GATResult
  GATContext_ServiceActions(GATContext context, GATTimePeriod_const timeout)
```

The first argument to this function is a `GATContext` which is used to broker any required resources while the second argument is a `GATTimePeriod`. This `GATTimePeriod` indicates the amount of time that this function should gift the various functions processing the "event loop" of GAT. If the application programmer wishes to gift these various functions an unlimited amount of time the can pass `NULL` as the final argument to this function.

11.3 Some Useful Programs

11.3.1 gattop, A top for the Grid Age

As an example of using some of the functionality present in the monitoring package, we will create a top to top `top`, which we will baptise `gattop`. As one will recall `top` is a command line program which displays various information about a particular computer. For example, if I were to run the `top` command as follows

```
% top
```

I would obtain a screen looking something like

```
Processes:  49 total, 2 running, 47 sleeping... 134 threads            10:43:17
Load Avg:  0.96, 0.62, 0.54     CPU usage:  77.2% user, 22.0% sys, 0.8% idle
SharedLibs: num =  111, resident = 29.8M code, 3.28M data, 8.02M LinkEdit
MemRegions: num =  5759, resident =  115M + 11.6M private,  116M shared
PhysMem:  72.1M wired,  151M active,  312M inactive,  535M used,  488M free
VM: 3.08G + 80.0M   48792(0) pageins, 1(0) pageouts

  PID COMMAND      %CPU   TIME   #TH #PRTS #MREGS RPRVT  RSHRD  RSIZE  VSIZE
  961 top         10.3%  0:00.93   1    16     26  400K   412K   772K  27.1M
  948 vim          0.0%  0:00.13   1    12     29  612K  1.21M  1.53M  18.7M
  906 Mail         0.0%  0:53.93   4   139    247 13.6M  25.4M  28.3M   126M
  905 Safari       0.0%  3:35.04   6   110    383 37.1M  26.7M  47.9M   147M
  890 lookupd      0.0%  0:00.42   2    36     61  396K   980K  1.16M  28.5M
  784 iTunes       7.7% 23:47.46   9   207    232 12.4M  18.6M+ 21.1M   135M
  391 tcsh         0.0%  0:00.58   1    13     21  408K   644K   956K  22.1M
  390 login        0.0%  0:00.03   1    13     37  144K   404K   500K  26.9M
  389 tcsh         0.0%  0:00.27   1    13     21  404K   644K   944K  22.1M
  388 login        0.0%  0:00.04   1    13     37  140K   404K   500K  26.9M
  387 Terminal    59.3%  3:39.97   4    70    176 2.38M- 12.0M+ 9.77M+  130M
  384 OmniGraffl   0.0%  0:06.34   3    85    212 5.02M  20.6M  13.2M   107M
  379 TeXShop      0.0% 54:26.04   2   113    371 9.81M  31.7M  33.8M   128M
  357 AppleSpell   0.0%  0:05.93   1    38     39  684K  1.97M  2.17M  36.3M
  350 automount    0.0%  0:00.03   2    29     27  224K   888K   924K  28.3M
  347 automount    0.0%  0:00.10   2    28     27  224K   888K   948K  28.3M
  344 rpc.lockd    0.0%  0:00.00   1     9     16   84K   364K   140K  17.7M
  335 nfsiod       0.0%  0:00.00   5    29     23  100K   316K   156K  19.6M
  320 ntpd         0.0%  0:03.75   1    10     18  132K   520K   336K  17.9M
  279 iCalAlarmS   0.0%  0:01.55   1    60     69  616K  2.46M  2.43M  84.8M
  276 Finder       0.0%  0:06.81   1   101    146 5.16M  11.0M  9.43M   107M
  275 SystemUISe   0.0%  1:25.18   2   185    209 2.35M  9.04M  6.89M   130M
  274 Dock         0.0%  0:13.48   2   102    119 1.04M  10.7M  3.65M  94.1M
  270 pbs          0.0%  0:01.20   2    32     47  616K  1.50M  1.88M  43.9M
  259 diskimages   0.0%  1:54.58   4    60     75 1.15M  2.08M  2.60M  54.4M
  240 cupsd        0.0%  0:03.42   1    11     28  392K   512K   668K  27.9M
  219 crashrepor   0.0%  0:00.01   1    17     19  120K   324K   172K  26.7M
  186 DirectoryS   0.0%  0:00.60   2    55     90  560K  2.52M  2.35M  30.5M
  182 loginwindo   0.0%  0:06.13   6   187    194 3.25M  9.42M  7.41M  88.4M
```

178	ATSServer	0.0%	0:19.92	2	72	268	1.66M	14.8M	5.48M	108M
176	ioupsd	0.0%	0:00.02	1	19	18	108K	324K	384K	26.7M
175	WindowServ	7.7%	15:49.40	2	222	1055	13.4M	39.3M+	48.1M+	115M
170	SecuritySe	0.0%	0:00.55	1	77	29	484K	1.16M	1.21M	28.2M
158	distnoted	0.0%	0:01.38	1	35	20	232K	736K	704K	27.1M
157	mDNSRespon	0.0%	0:00.51	2	33	31	268K	852K	844K	27.3M
154	coreservic	0.0%	0:03.54	1	70	158	1.79M	12.2M	6.06M	34.5M
153	cron	0.0%	0:00.17	1	10	21	104K	348K	188K	27.0M
148	KernelEven	0.0%	0:00.02	1	10	17	84K	328K	128K	26.7M
122	dynamic_pa	0.0%	0:00.00	1	12	16	72K	320K	124K	17.7M
119	update	0.0%	0:07.33	1	9	15	80K	316K	120K	17.6M
117	netinfod	0.0%	0:00.76	1	10	24	164K	456K	340K	26.8M
93	notifyd	0.0%	0:00.49	2	68	25	164K	352K	264K	18.2M
88	diskarbitr	0.0%	0:02.71	1	149	27	680K	836K	1.26M	27.2M
87	configd	0.0%	3:29.30	3	135	157	596K	1.64M	1.26M	29.4M
85	kextd	0.0%	0:03.75	2	17	22	1.21M	764K	1.17M	28.7M
79	syslogd	0.0%	0:00.33	1	9	16	96K	344K	208K	17.7M
2	mach_init	0.0%	0:00.90	2	168	17	128K	340K	212K	18.2M
1	init	0.0%	0:00.06	1	12	15	72K	328K	300K	17.6M
0	kernel_tas	0.6%	6:10.68	37	2	809	6.21M	0K	56.2M	727M

and this display of information would be updated every second or so. We are going to create a similar program which, instead of monitoring a particular computer, monitors a set of computers on the grid. In particular, if we call `gattop` as follows

```
$ gattop gridlab.org File.hrd
```

then we would be instructing `gattop` to monitor all resources in the virtual organization `gridlab.org` which are described by the hardware resources specified in the GATRL file `File.hrd`. If, for example, one wanted to monitor all machines in the `gridlab.org` virtual organization with at least a one GB of memory, one would call `gattop` with the following syntax

```
$ gattop gridlab.org File.hrd
```

where the file `File.hrd` looked like

```
memory.size=1
```

As to what this `gattop` prints out upon monitoring a hardware resource, it looks something like this

```
host = xeon01.aei-potsdam.mpg.de
  event time = 3284328432908
  event value = 468145
  metric name = host.mem.free
  metric unit = KiB
  metric parameters
    host = xeon01.aei-potsdam.mpg.de

host = xeon02.aei-potsdam.mpg.de
```

```
  event time = 3284328432908
  event value = 32448
  metric name = host.net.total.byte
  metric unit = Byte
  metric parameters
    host = xeon02.aei-potsdam.mpg.de
    interface = eth0

 ...
```

Lets look at what this print out means by looking at the first grouping of lines

```
host = xeon01.aei-potsdam.mpg.de
  event time = 3284328432908
  event value = 468145
  metric name = host.mem.free
  metric unit = KiB
  metric parameters
    host = xeon01.aei-potsdam.mpg.de
```

The information here is rather obvious. This is printing out the result of an `GATMetricEvent` being fired to our `gattop` application. It indicates that the host from where this information is coming is called "xeon01.aei-potsdam.mpg.de." The corresponding sensor measurement occurred 3284328432908 seconds after the epoch and the sensor's measured value is 468145. The name of the metric is "host.mem.free" and the value reported by the sensor is measured in "KiB." Finally, the metric has a single parameter "host" with value "xeon01.aei-potsdam.mpg.de."

That's basically all there is to `gattop`. The entire man page is as follows

```
NAME
    gattop - monitors hardware resources on a grid

SYNOPSIS
    gattop vo File.hrd

DESCRIPTION
    The command line utility gattop provides an on-going look
    at hardware resources on a grid in real time. It displays
    a listing of the output of various sensors planted within
    the the specified hardware resources.

    In addition this utility introduces a new specification,
    as if there were a dearth of them, called GATRL, which
    aims to be the most simple specification of a hardware
    resource description known to man.

    A GATRL file is simply a set of name/value pairs separated
    by an "=" sign

    name=value
```

```
    Each name/value pair occupies a single line in a GATRL file.
    For example, to specify the name value pairs size=big and
    color=red and a GATRL file would contain the following lines

    size=big
    color=red

    The motto for GATRL is "GATRL it's not rocket science."

    Using the simple GATRL file format introduced above one can
    specify a hardware resource description. One simply uses the
    corresponding supported name/value pairs in the GATRL file.

    For example, if I wanted to specify a hardware resource
    description using a GATRL file I might write something like
    this

    memory.size=1024
    machine.type=Power Macintosh
    cpu.type=powerpc

    Note that in a hardware resource description the value corresponding
    to the name "memory.size" is a Float the utility gatrun takes care to
    make sure that the supported names have values of the apropos type.
    All other values are treated as strings.

 EXAMPLES
    The following shows how to monitor a hardware resource described
    in the GATRL file ape.hrd within the virtual organization ape.org

      gattop ape.org ape.hrd

DIAGNOSTICS
     The gattop utility exits 0 on success, and >0 if an error occurs.

COMPATIBILITY
     The gattop utility is compatible with blondes who like taking
     long walks on the beach and cozying up in-front of the fireplace
     before a long night of Celebrity Death Match.

SEE ALSO
     gatbottom(1), gatmiddle(1), gatfujiya(1)

STANDARDS
     The gattop utility conforms to its own standards of hygiene and good
     grooming, not to mention the fact that it has the strongest of morals
     and would never under any circumstances kiss on the first date.
```

```
HISTORY
     The gattop command sprung forth fully formed from the ear of Tom
     Thumb, it was perfect in all its limbs, but no longer than Tom
     Thumb's thumb.

BUGS
     gatop annoys all those committed to the reunification of Gondwana
     due to its constant monitoring of their activities which violate
     articles 1 and 55 of the UN charter.
```

Ok, now for the code...

```
#include <stdio.h>
#include <string.h>

#include "GAT.h"


static void GATTop_PrintUsage( void );
static GATResult GATTop_Run( int argc, char *argv[] );
static GATResult GATTop_PrintHost( GATMetricEvent event );
static GATResult GATTop_PrintTime( GATMetricEvent event );
static GATResult GATTop_PrintValue( GATMetricEvent event );
static GATResult GATTop_PrintMetricUnit( GATMetricEvent event );
static GATResult GATTop_PrintMetricName( GATMetricEvent event );
static GATResult GATTop_RegisterMetricListener( int argc, char *argv[] );
static GATResult GATTop_PrintMetricParameters( GATMetricEvent event );
static GATResult GATTop_ParseGATRL( const char *file, GATTable *table );
static GATResult GATTop_MetricListener( void *data, GATMetricEvent event);
static  GATResult GATTop_GATTable_AddFloat( GATTable table, const char *name, const char *va

int main( int argc, char *argv[] )
{
  int returnValue;
  GATResult result;

  /* Check Command Line Arguments */
  if( argc != 3 )
  {
    /* Print Usage */
    GATTop_PrintUsage();

    /* Set result */
    result = GAT_INVALID_PARAMETER;
  }
  else
  {
    /* Run gattop */
    result = GATTop_Run( argc, argv );
```

```
  }

  /* Set returnValue */
  if( GAT_SUCCEEDED( result ) )
  {
    returnValue = 0;
  }
  else
  {
    returnValue = 1;
  }

  /* Return to OS */
  return returnValue;
}

/**
 * Prints the usage for gattop
 */
 static void GATTop_PrintUsage( void )
 {
   printf("NAME\n");
   printf("    gattop - monitors hardware resources on a grid \n");
   printf("\n");
   printf("SYNOPSIS\n");
   printf("    gattop vo File.hrd \n");
   printf("\n");
   printf("DESCRIPTION\n");
   printf("    The command line utility gattop provides an on-going look\n");
   printf("    at hardware resources on a grid in real time. It displays\n");
   printf("    a listing of the output of various sensors planted within\n");
   printf("    the the specified hardware resources.\n");
   printf("\n");
   printf("    In addition this utility introduces a new specification, \n");
   printf("    as if there were a dearth of them, called GATRL, which \n");
   printf("    aims to be the most simple specification of a hardware \n");
   printf("    resource description known to man. \n");
   printf("\n");
   printf("    A GATRL file is simply a set of name/value pairs separated \n");
   printf("    by an "=" sign\n");
   printf("\n");
   printf("    name=value\n");
   printf("\n");
   printf("    Each name/value pair occupies a single line in a GATRL file. \n");
   printf("    For example, to specify the name value pairs size=big and \n");
   printf("    color=red and a GATRL file would contain the following lines\n");
   printf("\n");
   printf("    size=big\n");
   printf("    color=red\n");
```

```
printf("\n");
printf("    The motto for GATRL is "GATRL it's not rocket science."\n");
printf("\n");
printf("    Using the simple GATRL file format introduced above one can \n");
printf("    specify a hardware resource description. One simply uses the \n");
printf("    corresponding supported name/value pairs in the GATRL file.\n");
printf("\n");
printf("    For example, if I wanted to specify a hardware resource  \n");
printf("    description using a GATRL file I might write something like \n");
printf("    this\n");
printf("\n");
printf("    memory.size=1024\n");
printf("    machine.type=Power Macintosh\n");
printf("    cpu.type=powerpc\n");
printf("    \n");
printf("    Note that in a hardware resource description the value corresponding\n");
printf("    to the name "memory.size" is a Float the utility gatrun takes care to\n");
printf("    make sure that the supported names have values of the apropos type.\n");
printf("    All other values are treated as strings. \n");
printf("\n");
printf(" EXAMPLES\n");
printf("    The following shows how to monitor a hardware resource described\n");
printf("    in the GATRL file ape.hrd within the virtual organization ape.org\n");
printf("\n");
printf("      gattop ape.org ape.hrd\n");
printf("\n");
printf("DIAGNOSTICS\n");
printf("     The gattop utility exists 0 on success, and >0 if an error occurs.\n");
printf("\n");
printf("COMPATIBILITY\n");
printf("     The gattop utility is compatible with blondes who like taking\n");
printf("     long walks on the beach and cozying up in-front of the fireplace\n");
printf("     before a long night of Celebrity Death Match.\n");
printf("\n");
printf("SEE ALSO\n");
printf("     gatbottom(1), gatmiddle(1), gatfujiya(1)\n");
printf("\n");
printf("STANDARDS\n");
printf("     The gattop utility conforms to its own standards of hygiene and good\n");
printf("     grooming, not to mention the fact that it has the strongest of morals \n");
printf("     and would never under any circumstances kiss on the first date.\n");
printf("\n");
printf("HISTORY\n");
printf("     The gattop command sprung forth fully formed from the ear of Tom\n");
printf("     Thumb, it was perfect in all its limbs, but no longer than Tom \n");
printf("     Thumb's thumb.\n");
printf("\n");
printf("BUGS\n");
printf("     gatop annoys all those committed to the reunification of Gondwana\n");
```

```
    printf("     due to its constant monitoring of their activities which violate\n");
    printf("     articles 1 and 55 of the UN charter.\n");

    /* Return to caller */
    return;
  }

/**
 * The entry point for this utility after the format of the command line
 * arguments has been checked.
 *
 * @param argc Number of command line arguments
 * @param argv Command line arguments
 * @return GATResult indicating completion status
 */
static GATResult GATTop_Run( int argc, char *argv[] )
{
  GATResult result;
  GATContext context;

  /* Register GATMetricListener */
  result = GATTop_RegisterMetricListener( argc, argv );

  /* Check Last Call */
  if( GAT_SUCCEEDED( result ) )
  {
    /* Initialize result */
    result = GAT_MEMORYFAILURE;

    /* Create GATContext */
    context = GATContext_Create();

    /* Check GATContext Creation */
    if( NULL != context )
    {
      /* Initialize result */
      result = GAT_SUCCESS;

      /* Yield to Event Processing Code */
      while( GAT_SUCCEEDED( result ) )
      {
        /* Service Actions */
        result = GATContext_ServiceActions( context, NULL );
      }

      /* Destroy GATContext */
      GATContext_Destroy( &context );
    }
  }
```

```
  /* Return to caller */
  return result;
}


/**
 * This function registers the GATMetricListener whcih will print out
 * the various GATMetricEvents
 *
 * @param argc Number of command line arguments
 * @param argv Command line arguments
 * @return GATResult indicating completion status
 */
static GATResult GATTop_RegisterMetricListener( int argc, char *argv[] )
{
  GATString vo;
  GATTable table;
  GATResult result;
  GATObject object;
  GATMetric *metric;
  GATuint32 cookies;
  GATContext context;
  GATResource *resource;
  GATResourceBroker broker;
  GATList_GATMetric metrics;
  GATList_GATResource resources;
  GATList_GATMetric_Iterator endMetric;
  GATList_GATMetric_Iterator currentMetric;
  GATList_GATResource_Iterator endResource;
  GATHardwareResourceDescription description ;
  GATList_GATResource_Iterator currentResource;

  /* Initialize result */
  result = GAT_MEMORYFAILURE;

  /* Create GATContext */
  context = GATContext_Create();

  /* Check GATContext Creation */
  if( NULL != context )
  {
    /* Create GATString */
    vo = GATString_Create( argv[1], strlen( argv[1] ) + 1, "ASCII" );

    /* Check GATString Creation */
    if( NULL != vo )
    {
      /* Create GATResourceBroker */
```

```
broker = GATResourceBroker_Create( context, NULL, vo );

/* Check GATResourceBroker Creation */
if( NULL != broker )
{
  /* Parse GATRL File */
  result = GATTop_ParseGATRL( argv[2], &table );

  /* Check Last Call */
  if( GAT_SUCCEEDED( result ) )
  {
    /* Initialize result */
    result = GAT_MEMORYFAILURE;

    /* Create GATHardwareResourceDescription */
    description = GATHardwareResourceDescription_Create( table );

    /* Check GATHardwareResourceDescription Creation */
    if( NULL != description )
    {
      /* Find Resources */
      result = GATResourceBroker_FindResources( broker, description, &resources );

      /* Check Last Call */
      if( GAT_SUCCEEDED( result ) )
      {
        /* Obtain Current Iterator */
        currentResource = GATList_GATResource_Begin( resources );

        /* Check Last Call */
        if( NULL != currentResource )
        {
          /* Obtain End Iterator */
          endResource = GATList_GATResource_End( resources );

          /* Check Last Call */
          if( NULL != endResource )
          {
            /* Loop Over resources */
            while( (NULL != currentResource) && (currentResource != endResource) )
            {
                /* Obtain Current GATResource */
                resource = GATList_GATResource_Get( currentResource );

                /* Check Last Call */
                if( NULL != resource )
                {
                  /* Convert resource to a GATObject */
                  object = GATResource_ToGATObject( *resource );
```

```
        /* Obtain Metrics */
        result = GATMonitorable_GetMetrics( object, &metrics );

        /* Check Success of Last Call */
        if( GAT_SUCCEEDED( result ) )
        {
          /* Obtain Current Iterator */
          currentMetric = GATList_GATMetrics_Begin( metrics );

          /* Check Last Call */
          if( NULL != currentMetric )
          {
            /* Obtain End Iterator */
            endMetric = GATList_GATMetrics_End( metrics );

            /* Check Last Call */
            if( NULL != endMetric )
            {
              /* Loop Over Metrics */
              while( (NULL != currentMetric)          &&
                          (GAT_SUCCEEDED(result)) &&
                          (currentMetric != endMetric)
                        )
              {
                /* Obtain Current GATMetric */
                metric = GATList_GATMetrics_Get( currentMetric );

                /* Check Last Call */
                if( NULL != metric )
                {
                  /* Add Listener */
                  result = GATMonitorable_AddMetricListener( object, GATTop_
                }

                /* Increment Current */
                currentMetric = GATList_GATMetrics_Next( currentMetric );
              }
            }
          }

          /* Destroy metrics */
          GATList_GATMetric( &metrics );
        }
      }

      /* Increment Current */
    currentResource = GATList_GATResource_Next( currentResource );
  }
```

```
              }
            }

            /* Destroy GATList_GATResource */
            GATList_GATResource_Destroy( &resources );
          }

          /* Destroy GATHardwareResourceDescription */
          GATHardwareResourceDescription_Destroy( &description );
        }
      }

      /* Destroy GATResourceBroker */
      GATResourceBroker_Destroy( &broker );
    }

    /* Destroy GATString */
    GATString_Destroy( &vo );
  }

  /* Destroy GATContext */
  GATContext_Destroy( &context );
  }

  /* Return to Caller */
  return result;
}

/**
 * Parses the specified GATRL file into a set of name/value pairs and places
 * those name value pairs in the passed GATTable
 *
 * @param fileName The specified GATRL file
 * @param table The GATTable into which names/values are parsed.
 * @return GATResult indicating completion status
 */
static GATResult GATTop_ParseGATRL( const char *fileName, GATTable *table )
{
  FILE *file;
  char *value;
  char *name;
  GATResult result;
  char nextLine[2048];

  /* Assume Invalid Parameter */
  result = GAT_INVALID_PARAMETER;

  /* Check Parameter */
  if( NULL !=  table )
```

```
{
  /* Create GATTable */
  *table = GATTable_Create();

  /* Check GATTable Creation */
  if( NULL = (*table) )
  {
    /* Assume IO Error */
    result = GAT_IO_ERROR;

    /* Open file */
    file = fopen( fileName, "r" );

    /* Check Last Call */
    if( NULL == file )
    {
      /* Assume Invalid GATRL */
      result = GAT_UNKNOWN_FORMAT;

      /* Read Next Line */
      while( NULL != fgets( nextLine, 2048, file ) )
      {
        /* Read name */
        name = strtok( nextLine, "=" );

        /* Check Last Call */
        if( NULL != name )
        {
          /* Read value */
          value = strtok( NULL, "=" );

          /* Check Last Call */
          if( NULL != value )
          {
            /* Add name/values to table */
            if( 0 == strcmp( "memory.size", name ) )
            {
              /* Add memory.size, value is a float */
              result = GATTop_GATTable_AddFloat( *table, name, value );
            } else if( 0 == strcmp( "memory.accesstime", name ) )
            {
              /* Add memory.accesstime, value is a float */
              result = GATTop_GATTable_AddFloat( *table, name, value );
            } else if( 0 == strcmp( "memory.str", name ) )
            {
              /* Add memory.str, value is a float */
              result = GATTop_GATTable_AddFloat( *table, name, value );
            } else if( 0 == strcmp( "cpu.speed", name ) )
            {
```

```
                /* Add cpu.speed, value is a float */
                result = GATTop_GATTable_AddFloat( *table, name, value );
              } else if( 0 == strcmp( "disk.size", name ) )
              {
                /* Add disk.size, value is a float */
                result = GATTop_GATTable_AddFloat( *table, name, value );
              } else if( 0 == strcmp( "disk.accesstime", name ) )
              {
                /* Add disk.accesstime, value is a float */
                result = GATTop_GATTable_AddFloat( *table, name, value );
              } else if( 0 == strcmp( "disk.str", name ) )
              {
                /* Add disk.str, value is a float */
                result = GATTop_GATTable_AddFloat( *table, name, value );
              } else
              {
                /* Add name, value is a C String */
                result = GATTable_Add_String( *table, (const void *) name, value);
              }

              /* On Failure, break while Loop*/
              if( GAT_FAILED( result ) )
              {
                break;
              }
            }
          }
        }

        /* Close file */
        fclose( file );
      }
    }
  }

  /* Return to Caller */
  return result;
}

/**
 * Adds the name/value pair to the passed table interpreting the
 * value as a C String representation of a GATfloat32.
 *
 * @param table The GATTable which to augment
 * @param name The name of the name/value pair
 * @param value The value of the name/value pair
 * @return A GATResult indicating completion status
 */
static  GATResult GATTop_GATTable_AddFloat( GATTable table, const char *name, const char *va
```

```
{
  GATfloat32 floatValue;

  /Convert value to float */
  floatValue = (GATfloat32) atof( value );

  /* Add to table */
  return GATTable_Add_float( table, (const void *) name, floatValue );
}

/**
 * This function prints out information pertaining to the passed GATMetricEvent
 *
 * @param data The callback data
 * @param event The GATMetricEvent of interest
 * @return A GATResult indicating completion status
 */
static GATResult GATTop_MetricListener( void *data, GATMetricEvent event)
{
  GATResult result;

  /*Print Event Host */
  result = GATTop_PrintHost( event );

  /* Check Last Call */
  if( GAT_SUCCEEDED( result ) )
  {
    /* Print Event Time */
    result = GATTop_PrintTime( event );

    /* Check Last Call */
    if( GAT_SUCCEEDED( result ) )
    {
      /* Print Event Value */
      result = GATTop_PrintValue( event );

      /* Check Last Call */
      if( GAT_SUCCEEDED( result ) )
      {
        /* Print Metric Name */
        result = GATTop_PrintMetricName( event );

        /* Check Last Call */
        if( GAT_SUCCEEDED( result ) )
        {
          /* Print Metric Unit */
          result =  GATTop_PrintMetricUnit( event );

          /* Check Last Call */
```

```
          if( GAT_SUCCEEDED( result ) )
          {
            /* Print Metric Parameters Title */
            printf( "  metric parameters\n" );

            /* Print Metric Parameters */
            result = GATTop_PrintMetricParameters( event );
          }
        }
      }
    }
  }

  /* Return to Caller */
  return result;
}

/**
 * Prints the GATEvent's time
 *
 * @param event The GATMetricEvent of interest
 * @return A GATResult indicating completion status
 */
static GATResult GATTop_PrintTime( GATMetricEvent event )
{
  GATResult result;
  GATTime_const time;
  GATdouble64 timeValue;

  /* Initialize result */
  result = GAT_MEMORYFAILURE;

  /* Obtain GATTime */
  time = GATMetricEvent_GetEventTime( event );

  /* Check Last Call */
  if( NULL != time )
  {
    /* Set result */
    result = GAT_SUCCESS;

    /* Get GATTime Value */
    timeValue = GATTime_GetTime( time );

    /* Print timeValue */
    printf( "  event time = %f\n", timeValue );
  }

  /* Return to Caller */
```

```
  return result;
}

/**
 * Prints the GATMetricEvent's GATMetric name
 *
 * @param event The GATMetricEvent of interest
 * @return A GATResult indicating completion status
 */
static GATResult GATTop_PrintMetricName( GATMetricEvent event )
{
  GATResult result;
  const char *name;
  GATMetric_const metric;

  /* Initialize result */
  result = GAT_MEMORYFAILURE;

  /* Obtain Metric */
  metric = GATMetricEvent_GetMetric( event );

  /* Check Last Call */
  if( NULL != metric )
  {
    /* Obtain name */
    name = GATMetric_GetName( metric );

    /* Check Last Call */
    if( NULL != name )
    {
      /* Set result */
      result = GAT_SUCCESS;

      /* Print Metric Name */
      printf( "  metric name = %s\n", name );
    }
  }

  /* Return to Caller */
  return result;
}

/**
 * Prints the GATMetricEvent's GATMetric unit
 *
 * @param event The GATMetricEvent of interest
 * @return A GATResult indicating completion status
 */
static GATResult GATTop_PrintMetricUnit( GATMetricEvent event )
```

```
{
  const char *unit;
  GATResult result;
  GATMetric_const metric;

  /* Initialize result */
  result = GAT_MEMORYFAILURE;

  /* Obtain Metric */
  metric = GATMetricEvent_GetMetric( event );

  /* Check Last Call */
  if( NULL != metric )
  {
    /* Obtain unit */
    unit = GATMetric_GetUnit( metric );

    /* Check Last Call */
    if( NULL != unit )
    {
      /* Set result */
      result = GAT_SUCCESS;

      /* Print Metric Name */
      printf( "  metric unit = %s\n", unit );
    }
  }

  /* Return to Caller */
  return result;
}

/**
 * This function prints the value of the passed GATMetricEvent
 *
 * @param event The GATMetricEvent of interest
 * @return A GATResult indicating completion status
 */
static GATResult GATTop_PrintValue( GATMetricEvent event )
{
  GATType type;
  GATResult result;

  /* Set result */
  result = GAT_SUCCESS;

  /* Obtain GATType */
  type = GATMetricEvent_GetValueType( event );
```

```
/* Check for GATType_GATint16 Type */
if( GATType_GATint16 == type )
{
   GATint16 value;

   /* Get Value */
   result = GATMetricEvent_GetValue( event, (void *) value, 16 );

   /* Check Last Call */
   if( GAT_SUCCEEDED( result ) )
   {
     /* Print Value */
     printf( "  event value = %d\n", value );
   }
}

/* Check for GATType_GATint32 Type */
if( GATType_GATint32 == type )
{
   GATint32 value;

   /* Get Value */
   result = GATMetricEvent_GetValue( event, (void *) value, 32 );

   /* Check Last Call */
   if( GAT_SUCCEEDED( result ) )
   {
     /* Print Value */
     printf( "  event value = %d\n", value );
   }
}

/* Check for GATType_GATuint16 Type */
if( GATType_GATuint16 == type )
{
   GATuint16 value;

   /* Get Value */
   result = GATMetricEvent_GetValue( event, (void *) value, 16 );

   /* Check Last Call */
   if( GAT_SUCCEEDED( result ) )
   {
     /* Print Value */
     printf( "  event value = %d\n", value );
   }
}

/* Check for GATType_GATuint32 Type */
```

```
if( GATType_GATuint32 == type )
{
   GATuint32 value;

   /* Get Value */
   result = GATMetricEvent_GetValue( event, (void *) value, 32 );

   /* Check Last Call */
   if( GAT_SUCCEEDED( result ) )
   {
     /* Print Value */
     printf( "  event value = %d\n", value );
   }
}

/* Check for GATType_GATfloat32 Type */
if( GATType_GATfloat32 == type )
{
   GATfloat32 value;

   /* Get Value */
   result = GATMetricEvent_GetValue( event, (void *) value, 32 );

   /* Check Last Call */
   if( GAT_SUCCEEDED( result ) )
   {
     /* Print Value */
     printf( "  event value = %f\n", value );
   }
}

/* Check for GATType_GATdouble64 Type */
if( GATType_GATdouble64 == type )
{
   GATdouble64 value;

   /* Get Value */
   result = GATMetricEvent_GetValue( event, (void *) value, 64 );

   /* Check Last Call */
   if( GAT_SUCCEEDED( result ) )
   {
     /* Print Value */
     printf( "  event value = %f\n", value );
   }
}

/* Check for GATType_String Type */
if( GATType_String == type )
```

```
  {
     char value[2048];

     /* Get Value */
     result = GATMetricEvent_GetValue( event, (void *) value, 2048 );

     /* Check Last Call */
     if( GAT_SUCCEEDED( result ) )
     {
       /* Print Value */
       printf( "  event value = %s\n", value );
     }
  }

  /* All other values are not handled */

  /* Return to Caller */
  return result;
}

/**
 * Prints out the
 */
static GATResult GATTop_PrintMetricParameters( GATMetricEvent event )
{
  int count;
  void *keys[];
  GATType type;
  GATTable table;
  GATResult result;
  GATMetric_const metric;

  /* Initialize result */
  result = GAT_MEMORYFAILURE;

  /* Obtain GATMetric */
  metric = GATMetricEvent_GetMetric( event );

  /* Check Last Call */
  if( NULL != metric )
  {
    /* Obtain GATTable */
    result = GATMetric_GetParameters( metric, &table );

    /* Check Last Call */
    if( GAT_SUCCEEDED( result ) )
    {
      /* Obtain Keys */
      keys = GATTable_GetKeys( table );
```

```
/* Initialize count */
count = 0;

/* Loop Over Keys */
while( NULL != keys[count] )
{
  /* Print Name */
  printf( "    %s = ", (char *) keys[count] );

  /* Obtain Element Type */
  type = GATTable_Get_ElementType( keys[count] );

  /* Check for GATType_GATint16 Type */
  if( GATType_GATint16 == type )
  {
     GATint16 value;

     /* Get Value */
     result = GATTable_Get_short( table, keys[count], &value );

     /* Check Last Call */
     if( GAT_SUCCEEDED( result ) )
     {
       /* Print Value */
       printf( "%d\n", value );
     }
  }

  /* Check for GATType_GATint32 Type */
  if( GATType_GATint32 == type )
  {
     GATint32 value;

     /* Get Value */
     result = GATTable_Get_int( table, keys[count], &value );

     /* Check Last Call */
     if( GAT_SUCCEEDED( result ) )
     {
       /* Print Value */
       printf( "%d\n", value );
     }
  }

  /* Check for GATType_GATdouble64 Type */
  if( GATType_GATdouble64 == type )
  {
     double value;
```

```
      /* Get Value */
      result = GATTable_Get_double( table, keys[count], &value );

      /* Check Last Call */
      if( GAT_SUCCEEDED( result ) )
      {
        /* Print Value */
        printf( "%f\n", value );
      }
    }

    /* Check for GATType_GATfloat32 Type */
    if( GATType_GATfloat32 == type )
    {
      float value;

      /* Get Value */
      result = GATTable_Get_float( table, keys[count], &value );

      /* Check Last Call */
      if( GAT_SUCCEEDED( result ) )
      {
        /* Print Value */
        printf( "%f\n", value );
      }
    }

    /* Check for GATType_String Type */
    if( GATType_String == type )
    {
      cahr value[2048];

      /* Get Value */
      result = GATTable_Get_String( table, keys[count], value, 2048 );

      /* Check Last Call */
      if( GAT_SUCCEEDED( result ) )
      {
        /* Print Value */
        printf( "%s\n", value );
      }
    }

    /* All other values are not handled */

    /* Increment count */
    count = count + 1;
  }
```

```
      /* Destroy keys */
      GATTable_ReleaseKeys( table, &keys);
    }

    /* Destroy GATTable */
    GATTable_Destroy( &table );
  }

  /* Return to Caller */
  return result;
}

/**
 *  This function prints the host from which the GATMetricEvent originated
 *
 * @param event The GATMetricEvent of interest
 * @return A GATResult indicating completion status
 */
static GATResult GATTop_PrintHost( GATMetricEvent event )
{
  char host[2048];
  GATResult result;
  GATObject_const object;
  GATTable_const hRDTable;
  GATHardwareResource_const hR;
  GATHardwareResourceDescription_const hRD;

  /* Initialize result */
  result = GAT_MEMORYFAILURE;

  /* Obtain GATObject Source */
  object = GATMetricEvent_GetSource( event );

  /* Check Last Call */
  if( NULL != object )
  {
    /* Convert GATObject_const to GATHardwareResource_const */
    hR = GATObject_ToGATHardwareResource_const( object );

    /* Obtain GATHardwareResourceDescription */
    result = GATHardwareResource_GetResourceDescription( hR, &hRD );

    /* Check Last Call */
    if( GAT_SUCCEEDED( result ) )
    {
      /* Initialize result */
      result = GAT_MEMORYFAILURE;
```

```
      /* Obtain Hardware Resource Description GATTable */
      hRDTable = GATHardwareResourceDescription_GetDescription( hRD );

      /* Check Last Call */
      if( NULL != hRDTable )
      {
        /* Obtain host */
        result = GATTable_Get_String( hRDTable, "machine.node", host, 2048 );

        /* Check Last Call */
        if( GAT_SUCCEEDED( result ) )
        {
          /* Print Host */
          printf("host = %s\n", host );
        }
      }
    }
  }

  /* Return to Caller */
  return result;
}
```

12 Event System

12.1 Being Spyed On: A User's Guide

Beyond spying on properly instrumented applications, GAT also allows for a reversal of roles in which the spy becomes the spied upon. Normally, at least in the world of international espionage, spies do not give permit for others to monitor their daily doings. But, in the world of things GAT, much is different. A GAT application can actually open it self up to be monitored by any party with the apropos rights to do so.

As an example of why this might be advantageous, consider the case of an application which spins off child applications, each of which tackles a small portion of the task the large application is yoked with. Each of these child applications might need to communicate with the parent application to provide any number of application specific datum – its general health, iteration number, preliminary computation results, requests for a new data sets, and on and on. These datum could be used by the parent application to determine how things are going with its children. Has this child process crashed? Is this child at iteration 100? Has this child gotten to the second half of the computation? Does this child need more data to process? The event system makes it possible for the child processes to be monitored so as to report arbitrary application specific data to their parent.

The GAT event system not only allows applications to be monitored; the event system allows masochistic applications to open themselves up to being commanded about by sadistic external processes. Beyond any perverse gratification that might be gleaned from the act of being commanded about, this act of allowing an external application to command a GAT application is actually extremely useful.

If we continue the arc of the previous example, then its actually easy to illustrate the utility of a GAT application being bossed about from a process external to itself. Consider for example the various child processes introduced previously. The parent may monitor any number of characteristics of these child processes. For example, each of these child processes may decide to allows themselves to be monitored for something as simple as if the machine they are running on is about to be shut down[10]. The parent process can then query a child to determine if the machine on which it is running is about to go down. If it finds one of its children is stranded on a host that's about to be blinked out of existence, then what is it to do?

[10]Often when a machine is about to be shut down it will broadcast a message indicating that in 5 minutes, or so, it will go down.

This is where the command portion of the event system shines. The command portion of the event system allows an application to open itself up to be commanded about by other processes. In particular, it allows for an application to be checkpointed by an external process. So, this parent process, in fear of its child's imminent demise, can command the child to checkpoint itself. If the child has instrumented itself to be checkpointed – and checkpoint instrumentation is oh so simple with GAT – the child will simply checkpoint itself and avert disaster.

12.2 The Event Package

The event package is one of the smallest packages within all of GAT; size, however, isn't everything. The event package only contains three new elements, the function pointer type `GATRequestListener`, the function pointer type `GATRequestNotifier`, and the enumeration `GATRequestType`. Beyond these new elements the event package makes use only of elements that are part of other GAT packages. Lets examine conceptually the details of each of these new elements.

The first new element, the function pointer type `GATRequestListener`, allows the spied upon to be called upon by the spies or it allows a specific command to be carried out. In particular, when an application uses the event package, its main purpose in doing so is to either be monitored by a remote process or to respond to a command sent by some remote process. Both of these are accomplished through functions of the type `GATRequestListener`.

A GAT application which wishes to be spied upon or respond to external commands registers with GAT a function with a signature matching that of the type `GATRequestListener`. Upon the external process contacting the GAT application, either for monitoring information or to issue a command, this registered function is called in response to the external process's query. It is through the implementation of this function that a GAT application can respond to external requests.

The second new element in the GAT event system is the function pointer `GATRequestNotifier`. When a registered `GATRequestListener` is called as a result of a query from an external process, the `GATRequestListener` is passed a `GATRequestNotifier`. It is through this `GATRequestNotifier` that a GAT application can pass information back to GAT and thus to the calling remote process. In particular, upon completing whatever processing needs to occur, a GAT application calls the `GATRequestNotifier` passed to its registered `GATRequestListener` and passes whatever information is needed to this `GATRequestNotifier`.

The third new element in the GAT event system is much more pedestrian than the function pointer types we covered above; it is the enumeration `GATRequestType`. When registering a `GATRequestListener` with the GAT engine, a GAT application must specify if the registered `GATRequestListener` is for command requests or informational monitoring requests. This enumeration allows one to specify this one bit of information. The enumeration `GATRequestType` can simply take one of two values. The first specifies that a command `GATRequestListener` is being registered. The second specifies that a informational or monitoring `GATRequestListener` is being registered. Elementary my dear Watson.

12.2.1 Adding and Removing RequestListeners

Before a GAT application can allow itself to be monitored or commanded about it must register its ability to do so with GAT. This is accomplished through use of the following function

```
GATResult
  GATSelf_AddRequestListener(GATContext context,
    GATPreferences_const request_prefs, GATRequestListener listener,
    void *data, GATRequestType type, GATTable_const parameters,
    const char *name, GATuint32 *cookie)
```

This function returns a `GATResult` which indicates its completion status, the type `GATResult` is covered in Appendix C. The first argument is a `GATContext` instance which is used to broker resources, as is usual. The next argument is a `GATPreferences` instance which, as usual, is used to select the proper adaptor. The next argument is the `GATRequestListener` which is to be registered. The following argument is a `void *` pointer which points to data which should be passed back to the registered `GATRequestListener` when it is called. The next argument is an enumeration of type `GATRequestType`. This type can currently take on one one of two values `GATRequestType_Command`, for a `GATRequestListener` which can deal with command requests, or `GATRequestType_Information` for a `GATRequestListener` that can deal with monitoring requests.

The next argument takes some explaining. It is a `GATTable` instance, a type covered in Appendix F. For the case of a command request type this table can be completely empty. However, for the case of a informational/monitoring request listener this table contains the information which describes the new metric which is able to be monitored. In particular this table has to contain the following key/value pairs

Key	Value Type	Description
`Metric parameters`	`GATTable`	The metric's parameters.
`Metric measurement type`	`GATMeasurementType`	The metric's measurement type.
`Metric data type`	`GATType`	An indicator of the measured type
`Metric unit`	`char *`	A String indicator of the measured value's unit.

Table 10: The key/values pairs defining a GATMetric.

This table allows one to define all the various values obtainable but examining a `GATMetric` as described in section 11.2.2 mod one, the name of the `GATMetric`, which is described by the next argument to `GATSelf_AddRequestListener` a `const char *`. In the case of a command request listener this "name" argument contains the name of the command. Currently the only name supported by GAT is "checkpoint".

The final argument to this function is a `GATuint32 *` pointer. It is through this pointer that GAT returns a `GATuint32`. This `GATuint32` is a unique identifier, a "cookie" if you will, used by GAT and a GAT application to keep track of this registered function. For example, this `GATuint32` can be used by a GAT application to remove a registered `GATRequestListener` with the function

```
GATResult
  GATSelf_RemoveRequestListener(GATContext context, GATuint32 cookie)
```

As is usual, this function returns a `GATResult`, a type covered in Appendix C, indicating its completion status. The first argument is a `GATContext` instance used to broker resources for this call and the final argument is the "cookie" which identifies uniquely the registered `GATRequestListener` that one wishes to un-register.

12.2.2 RequestListener

Now, as the type `GATRequestListener` is simply a function pointer, one can't really create one in the sense of normal GAT class instance. But a GAT application has to contain its own function which has the signature defined by the function pointer type `GATRequestListener`. ("I can only show you the door. You're the one that has to walk through it.")

The type `GATRequestListener` is defined as follows

```
typedef GATResult (*GATRequestListener)(void *, GATRequestNotifier, void *);
```

As is usual, this function returns a `GATResult` indicating its completion status, the type `GATResult` is covered in Appendix C. The first argument to this function is a `void *` pointer. When a registered `GATRequestListener` is called this `void *` pointer points to the data passed during the function's registration, in particular the fourth argument to the registration function `GATSelf_AddRequestListener` when the `GATRequestListener` was registered. The next argument to this function is a `GATRequestNotifier` which the GAT application uses to pass information back to GAT and, as a result, to the calling process. We will cover `GATRequestNotifier` in detail below. The final argument to this function is another `void *` pointer. This pointer points to data which must be passed back to GAT when a `GATRequestListener` is finished processing.

12.2.3 RequestNotifier

As mentioned previously, the type `RequestNotifier` is a function pointer used by a GAT application to return information to GAT and in turn to the remote process making a request. This function type is defined as follows

```
typedef GATResult (*GATRequestNotifier)(void *, GATTable_const);
```

It returns a `GATResult` indicating the completion status of the function, the type `GATResult` is covered in Appendix C. Its first argument is a `void *` which points to the data passed as the final argument to the `GATRequestListener` with which this function is associated. The final argument is a `GATTable` through which this function returns information to GAT and thus to the remote, requesting process. The actual information contained in this table is dependent upon the type of the request.

For example, in the case of an a `GATRequestListener` of type `GATRequestType_Information`, this `GATTable` must contain the following set of key/value pairs.
For other `GATRequestListener` types the information passed back in this table is different. For example for a `GATRequestListener` of type `GATRequestType_Command` which has the name "checkpoint" must return a table
Both of these arguments are optional and are formulated as follows. Each is a sequence of strings which are quoted and comma-separated. Each of these quoted strings is formatted as if it were to

Key	Value Type	Description
`Metric value`	`Object`	A GATObject giving metric's value.
`Metric timestamp`	`GATTime`	A GATTime indicating information's harvest time.

Table 11: Information Requests: The key/values pairs defining a GATMetricEvent.

Key	Value Type	Description
`Checkpoint Files value`	`String`	A GATString indicating the checkpoint files.
`Restart Files`	`String`	A GATString indicating the restart files.

Table 12: Checkpoint Requests: The key/values pairs defining a checkpoint response.

be passed to the `GATLocation` constructor. In future other types of `GATRequestType_Command` will be added beyond the "command" type and associated with each new type will be an associated set of data to be included in the `GATTable`.

12.3 Some Useful Programs

So, now you're big boys and girls in the world of things GAT. You've seen everything that GAT has to offer, from the `GATAdvertService` to `GATResult`; so, we leave the example programs for this chapter as an exercise to the reader. Write your own!

A Appendix: GATTypes

The full catalog of values which GATType can take on is given as follows

GATType_GATint16	GATType_GATint32
GATType_GATuint16	GATType_GATuint32
GATType_GATfloat32	GATType_GATdouble64
GATType_String	GATType_PlainOldData
GATType_GATObject	GATType_GATStatus
GATType_GATList	GATType_GATTable
GATType_GATList	GATType_GATTable
GATType_GATResourceDescription	GATType_GATSoftwareResourceDescription
GATType_GATHardwareResourceDescription	GATType_GATFile
GATType_GATLogicalFile	GATType_GATLocation
GATType_GATTime	GATType_GATTimePeriod
GATType_GATSoftwareResourceDescription	GATType_GATPreferences
GATType_GATString	GATType_GATResourceBroker
GATType_GATResource	GATType_GATSoftwareResource
GATType_GATHardwareResource	GATType_GATReservation
GATType_GATJob	GATType_GATJobDescription
GATType_GATMemoryStream	GATType_GATMetric
GATType_GATMetricEvent	GATType_GATMonitorable_Impl
GATType_GATEndpoint	GATType_GATPipe
GATType_GATFileStream	GATType_GATSelf
GATType_GATRequest	GATType_GATRequestNotifier
GATType_GATContext	GATType_GATDistinguishedName
GATType_GATAdvertService	GATType_GATRegistry
GATType_NoType	

Table 13: The full catalog of values which GATType can take on.

B Appendix: GAT Primitive Types

The full dictionary of standard GAT primitive types

GAT Primitive Type	GAT Primitive Type Description
GATBool	Boolean with values `GATFalse` or `GATTrue`
GATint8	8 bit signed integer
GATuint8	8 bit un-signed integer
GATint16	16 bit signed integer
GATuint16	16 bit un-signed integer
GATint32	32 bit signed integer
GATuint32	32 bit un-signed integer
GATfloat32	32 bit floating point number
GATdouble64	64 bit double precision number

Table 14: The full dictionary of standard GAT primitive types.

In addition to the standard GAT primitive types, which are guaranteed to exist on any C89/C99 platform, there exist a set of optional GAT primitive types which may or may not exist on a given target platform. These optional GAT primitive types existence depends upon the 64 bit support which the platform provides. The full dictionary of such optional GAT primitive types is given by

GAT Primitive Type	GAT Primitive Type Description
GATint64	64 bit signed integer
GATuint64	64 bit un-signed integer

Table 15: The full dictionary of options GAT primitive types.

C Appendix: GATResult Codes

A variable of type `GATResult` is used to return the completion status of a function. This type is simply `typedef'd` to be a `GATint32`, a type covered in Appendix B,

```
/* The error/result code type */
typedef GATint32 GATResult;
```

The actual `GATint32` returned contained within a variable of type `GATResult` actually contains five pieces of information within this one integer value. These are shown in table 16.

Name	Description
Severity Code	A two bit number indicating how seriousness of the return value
Customer Code	A one bit number which mey be used by client code
Reserved Code	A one bit number which is reserved for internal use.
Facility Code	A 12 bit number indicating which faculty is reporting this result.
Facility Status Code	A 15 bit number indicating the faculty's return code.

Table 16: GATResult's component parts.

These various pieces of information are arranged in the `GATint32`, i.e. `GATResult`, as shown in figure 30,

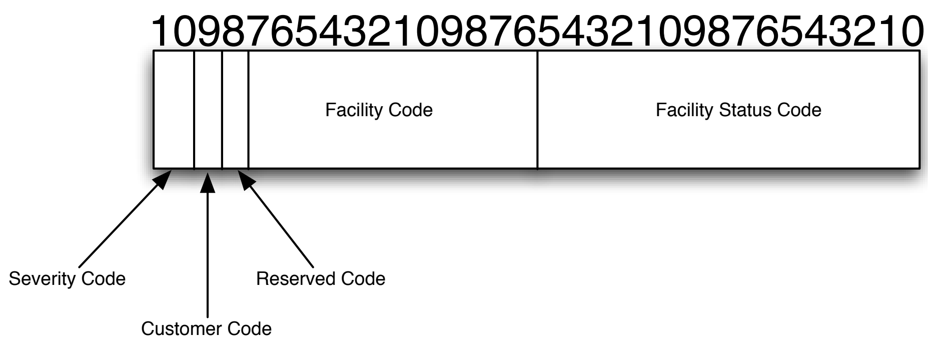

Figure 30: Layout of a GATResult in memory.

C.1 Severity Code

The Severity Code is a two bit number whose values have the semantics shown in table 17

Binary Value	Semantics
00	The GATResult indicates success.
01	The GATResult is informational.
10	The GATResult indicates a warning.
11	The GATResult indicates an error.

Table 17: Severity Code's binary values and associated semantics.

C.2 Customer Code

The Customer Code is a one bit number which may be used by client code.

C.3 Reserved Code

The Reserved Code is a one bit number which is used internally by GAT.

C.4 Facility Code

GAT uses various third party facilities such as XDS, UUID, RegEx, SQLite... The Facility Code is a 12 bit number which indicates from which of these facilities a `GATResult` originated. The Facility Code's values are mapped to the various facilities as shown in table 18

Decimal Value	Facility
0	The GAT engine itself.
1	The XDS serlalization library.
2	The The UUID unique identifier library.
3	The RegEx regular expression library.
4	POSIX, the corresponding Facility Status Codes are errno based.
10	The GATFile CPI
11	The GATLogicalFile CPI
20	The SQLite database library.

Table 18: Facility Code's decimal values and the associated facilities.

C.5 Facility Status Code

The various values of the Facility Status Code are dependent upon the facility from which the corresponding `GATResult` originated. For that reason we will only examine here the various Facility Status Code's which arise at part of GAT. Their values and semantics are given by the table 19

Decimal Value	Semantics
1	The GAT engine experienced a generic failure.
2	The called for GAT engine functionality is not yet implemented.
3	The GAT engine experienced a memory related error.
4	The GAT engine experienced an error due to loading a duplicate adaptor.
5	The GAT engine failed to load an adaptor.
6	The corresponding GAT adaptor does not have a register function.
7	The GAT engine encountered a duplicate configuration.
8	The GAT engine encountered an error opening a file.
9	The GAT engine could not find the specifid key.
10	The GAT engine encountered an error associated with the key's type.
11	The GAT engine encountered due to an invalid handle.
12	The GAT engine encountered due to an invalid parameter.
13	The GAT engine encountered due to an unknown version.
14	The GAT engine has no registered CPI for the functionality.
15	The GAT engine encountered an error due to an invalid encoding.
16	The GAT engine encountered an error due to an incomplete encoding.
17	The GAT engine has no matching CPI for the functionality.
18	The GAT engine found no matching resource.
19	The GAT engine found no matching interface.
20	The GAT engine encounterd an unknown format.
21	The GAT engine encounterd an error due to a pre-existing key.
22	The GAT engine encounterd an error due to RPC failing.
23	The GAT engine encounterd an IO error.
24	The GAT engine encounterd an error due to an invalid state.
25	The GAT engine encounterd an error due to a type mismatch.
26	The GAT engine encounterd an error due to a buffer being to small.
27	The GAT engine encounterd an error due to an invalid object conversion.

Table 19: GAT's Facility Status Code's decimal values and associated semantics.

C.6 GATResult Macro's

Various parts of a `GATResult` can be accessed by a set of macros. In particular, there exist macros which obtain the Facility Status Code, Facility Code, and Severity Code. The macros and sematics are given by the following table 20

Macro	**Semantics**
`GAT_RESULT_CODE(rc)`	Returns the Facility Status Code of the passed `GATResult`.
`GAT_RESULT_FACILITY(rc)`	Returns the Facility Code of the passed `GATResult`.
`GAT_RESULT_SEVERITY(rc)`	Returns the Severity Code of the passed `GATResult`.

Table 20: Macros to obtain parts of a GATResult.

In addition to these macros there exist two macros which serve the more mundane role of determining if a particular `GATResult` corresponds to success or failure. These macros are detailed in the table 21.

Macro	**Semantics**
`GAT_SUCCEEDED(rc)`	Returns C "boolean" indicating if `rc` indicated success.
`GAT_FAILED(rc)`	Returns C "boolean" indicating if `rc` indicated failure.

Table 21: Macros to obtain GATResult success or failure.

D Appendix: GATPreferences

The majority of functionality present in GAT is implemented through the the auspices of software components called "adaptors." An adaptors is software component written by a third party which provide some functionality present in the GAT API. For example the class `GATFile` provides to the application programmer the functionality of moving files. GAT itself does not actually provide such functionality, but only delegates calls to move files to the appropriate adaptor. Hence, GAT, in all truth, simply provides a uniform interface to a set of adaptors.

GAT allows for various adaptors to provide the same functionality. So, for example, there may exist several adaptors which provide all the functionality present in the `GATFile` class. For example, one adaptor may communicate with remote computers using HTTP protocols while the second may use HTTPS. The question then arises how does the application programmer choose a particular adaptor or a particular class of adaptors to do her bidding? The answer to this question is `GATPreferences`.

The class `GATPreferences` can be thought of as a `GATObject` subclass which is a hashtable capable of holding a set of key/value pairs in which the keys and values are standard C strings, zero terminated a `char *`'s. When a particular adaptor is loaded by GAT it registers with GAT a `GATPreferences` describing itself. The application programmer, when she wishes to create a `GATFile`, say, passes the "Create" call a `GATPreferences` which describes, using key/value pairs, the adaptor the application programmer wishes to use. This application programmer provided `GATPreferences` is then "matched" against the adaptor provided `GATPreferences`, and the first adaptor with a "matching" `GATPreferences` instance is then used to do the application programmer's bidding.

This process of "matching" `GATPreferences` instances is actually relatively simple. For a `GATPreferences` instance `criteria` to match a second `GATPreferences` instance `preferences` all keys present in `criteria` must be present in `preferences` and their corresponding values must "match," a value in `preferences` is matched by a POSIX 1003.2 regular expression in `criteria`. GAT has abstracted this process of matching two `GATPreferences` instances in to a single call,

```
GATBool
  GATPreferences_Match( GATPreferences_const preferences,
                        GATPreferences_const criteria)
```

So, for example, to determine if a `GATPreferences` instance `criteria` matches a `GATPreferences` instance `preferences` we would proceed as follows

```
GATBool match;
GATPreferences criteria;
GATPreferences preferences;

criteria = ...
preferences = ...

match = GATPreferences_Match( preferences, criteria );

if( GATTrue == match )
{
  printf( "criteria matches preferences\n" );
}
else
{
  printf( "criteria does not match preferences\n" );
}
```

E Appendix: Regular Expressions

A regular expression (abbreviated as regexp or regex) is a string that describes a whole set of strings, according to certain syntax rules. These expressions are used by many text editors and utilities (especially in the Unix operating system) to search bodies of text for certain patterns and, for example, replace the found strings with a certain other string[11].

E.1 Brief History

The origin of regular expressions lies in automata theory and formal language theory (both part of theoretical computer science). These fields study models of computation (automata) and ways to describe and classify formal languages. A *formal language* is nothing but a set of strings. In the 1940s, Warren McCulloch and Walter Pitts described the nervous system by modelling neurons as small simple automata. The mathematician, Stephen Kleene, later described these models using his mathematical notation called regular sets. Ken Thompson built this notation into the editor qed, then into the Unix editor ed and eventually into grep. Ever since that time, regular expressions have been widely used in Unix and Unix-like utilities.

E.2 Regular Expressions in Formal Language Theory

Regular expressions consist of constants and operators that denote sets of strings and operations over these sets, respectively. Given a finite alphabet Σ the following constants are defined

- (*empty set*) $\emptyset$ denoting the set $\emptyset$
- (*empty string*) ϵ denoting the set $\{\epsilon\}$
- (*literal character*) $q \in \Sigma$ denoting the set $\{q\}$

and the following operations

- (*concatenation*) RS denoting the set $\{\alpha\beta \,|\, \alpha \in R \textit{ and } \beta \in S\}$. For example, $\{ab, c\}\{d, ef\} = \{abd, abef, cd, cef\}$.
- (*set union*) $R \cup S$ denoting the set union of R and S.

[11]This chapter is adapted from the Wikipedia regular expressions entry.

- (*Keeene star*) $R*$ denoting the smallest superset of R that contains ϵ and is closed under string concatenation. This is the set of all strings that can be made by concatenating zero or more strings in R. For example, $\{ab, c\}* = \{\epsilon, ab, c, abab, abc, cab, cc, ababab, \ldots\}$.

To avoid brackets it is assumed that the Kleene star has the highest priority, then concatenation and then set union. If there is no ambiguity then brackets may be omitted. For example, $(ab)c$ is written as abc and a $\cup (b(c*))$ can be written as a $\cup bc*$.

Sometimes the complement operator $\sim$ is added; $\sim R$ denotes the set of all strings over Σ that are not in R. In that case the resulting operators form a Kleene algebra. The complement operator is redundant: it can always be expressed by only using the other operators.

Examples:

- A $\cup b*$ denotes $\{a, \epsilon, b, bb, bbb, \ldots\}$
- $(a \cup b)*$ denotes the set of all strings consisting of a's and b's, including the empty string
- $b * (ab*)*$ the same
- $ab * (c \cup \epsilon)$ denotes the set of strings starting with a, then zero or more b's and finally optionally a c.
- $(bb \cup a(bb) * aa \cup a(bb) * (ab \cup ba)(bb) * (ab \cup ba))*$ denotes the set of all strings which contain an even number of b's and a number of a's divisible by three.

Regular expressions in this sense can express exactly the class of languages accepted by finite state automata, the regular languages. There is, however, a significant difference in compactness. Some classes of regular languages can only be described by automata that grow exponentially in size, while the required regular expressions only grow linearly. Regular expressions correspond to the type 3 grammars of the Chomsky hierarchy and may be used to describe a regular language.

We can also study expressive power within the formalism. As the example shows, different regular expressions can express the same language, the formalism is redundant.

It is possible to write an algorithm which given two regular expressions decides whether the described languages are equal - essentially, it reduces each expression to a minimal deterministic finite state automaton and determines whether they are equivalent.

To what extent can this redundancy be eliminated? Can we find an interesting subset of regular expressions that is still fully expressive? Kleene star and set union are obviously required, but perhaps we can restrict their use. This turns out to be a surprisingly difficult problem. As simple as the regular expressions are, it turns out there is no method to systematically rewrite them to some normal form. They are not finitely axiomatizable. So, we have to resort to other methods. This leads to the star height problem.

E.3 POSIX Regular Expression Syntax

In this syntax, most characters are treated as literals - they match only themselves (a matches a, abc matches bc, etc). The exceptions are called *metacharacters*:

- `.` Matches any single character
- `[ ]` Matches a single character that is contained within the brackets - $[abc]$ matches a, b, or c. $[a-z]$ matches any lowercase letter.
- `[^]` Matches a single character that is not contained within the brackets - $[\hat{}a-z]$ matches any single character that isn't a lowercase letter
- `^` Matches the start of the line
- `$` Matches the end of the line
- `( )` Mark a part of the expression. What the enclosed expression matched to can be recalled by `\n` where n is a digit from 1 to 9.
- `\n` Where n is a digit from 1 to 9; matches to the exact string what the expression enclosed in the nth left parenthesis and its pairing right parenthesis has been matched to. This construct is theoretically irregular and has not adopted in the extended regular expression syntax.
- `*` A single character expression followed by $*$ matches to zero or more iteration of the expression. For example, $[xyz]*$ matches to ϵ, x, y, zx, zyx, and so on. A $\backslash n*$, where n is a digit from 1 to 9, matches to zero or more iterations of the exact string that the expression enclosed in the nth left parenthesis and its pairing right parenthesis has been matched to. For example, $(a??)\backslash 1$ matches to $abcbc$ and $adede$ but not $abcde$. An expression enclosed in (and) followed by $*$ is deemed to be invalid. In some cases (e.g. /usr/bin/xpg4/grep of SunOS 5.8), it matches to zero or more iteration of the same string which the enclose expression matches to. In other some cases (e.g. /usr/bin/grep of SunOS 5.8), it matches to what the enclose expression matches to, followed by a literal $*$.
- `{x,y}` Match the last "block" at least x and not more than y times. - $a\{3,5\}$ matches aaa, $aaaa$ or $aaaaa$.
- `+` Match the last "block" one or more times - $ba+$ matches ba, baa, $baaa$ and so on
- `?` Match the last "block" zero or more times - $ba?$ matches b or ba
- `|` The choice (or set union) operator: match either the expression before or the expression after the operator - $abc|def$ matches abc or def.

Since the characters '(', ')', '[', ']', '.', '*', '?', '+', '^' and '$' are used as special symbols they have to be "escaped" somehow if they are meant literally. This is done by preceding them with '\' which therefore also has to be "escaped" this way if meant literally.

F Appendix: GATTable

The class `GATTable` is simply put a hashtable. It is a container for a set of key/value pairs and maintains an internal mapping between a given key and its corresponding value. A given key may not be associated with more than one value.

F.1 Creating and Destroying Table instances

One creates a `GATTable` instance through a call to the function

```
GATTable GATTable_Create(void)
```

This function returns a `GATTable` instance or `NULL` if it encounters some internal problem. The so created `GATTable` instance will contain no key/value pairs.

To destroy this so created `GATTable` instance on calls upon the function

```
void GATTable_Destroy(GATTable *table)
```

This function takes as its first argument a pointer to the `GATTable` instance to destroy. Upon completion this function frees any resources held by the passed `GATTable` instance.

F.2 Adding Key/Value Pairs

To add a key/value pair to a `GATTable` instance one uses one on the many "Add" functions. Each such "Add" function is able to add values of one given type, which can be either a primitive type or a GAT class. Generically such "Add" functions have the following format

```
GATResult GATTable_Add_Type(GATTable table, const void *key, Type value)
```

The first argument is the `GATTable` to which the key/value pair will be added. The second argument is the key, a standard C string. The final argument, and the function name itself, depends upon the particular type the function is able to add. For exmaple, it may be an `int`, `short`, `GATObject` . . . The full set of "Add" functions is as follows
All of these functions return a `GATResult`. The `GATResult`, covered in Appendix C, indicates their completion status.

GATTable_Add_int(GATTable table, const void *key, GATint32 data)
GATTable_Add_short(GATTable table, const void *key, GATint16 data)
GATTable_Add_double(GATTable table, const void *key, GATdouble64 data)
GATTable_Add_float(GATTable table, const void *key, GATfloat32 data)
GATTable_Add_String(GATTable table, const void *key, const char *data)
GATTable_Add_GATObject(GATTable tbl, const void *key, GATObject_const obj)

Table 22: The full set of "Add" functions of GATTable.

F.3 Removing Key/Value Pairs

In contrast to the many functions required to add a key/value pair to a `GATTable`, there exists a single function to remove a given key/value pair. It is the following function

```
GATResult GATTable_Remove(GATTable table, const void *key)
```

The first argument to this function is the `GATTable` from which the key/value pair is to be removed. The second argument to this function is the of the key/value pair which is to be removed. Upon successful completion of this function, there will exist no key/value pair mapping in the passed `GATTable` instance corresponding to this passed key. The `GATResult`, covered in Appendix C, returned from this function indicates its completion status.

F.4 Obtaining Values Corresponding to a Given Key

To get a value from a `GATTable` instance corresponding to a given key one uses one on the many "Get" functions. Each such "Get" function is able to get values of a given type, which can be either a primitive type or a GAT class. Generically such "Get" functions have the following format

```
GATResult GATTable_Get_Type(GATTable table, const void *key, Type *value)
```

The first argument is the `GATTable` from which the value is to gotten. The second argument is the key, a standard C string. The final argument, and the function name itself, depends upon the particular type the function is able to get. For exmaple, it may be a pointer to an `int`, `short`, `GATObject` . . . The full set of "Get" functions is as follows

GATTable_Get_int(GATTable_const table, const void *key, GATint32 *data)
GATTable_Get_short(GATTable_const table, const void *key, GATint16 *data)
GATTable_Get_double(GATTable_const table, const void *key, GATdouble64 *data)
GATTable_Get_float(GATTable_const table, const void *key, GATfloat32 *data)
GATTable_Get_String(GATTable_const t, const void *k, const char *d, GATuint32 l))
GATTable_Get_GATObject(GATTable_const table, const void *key, GATObject *object)

Table 23: The full set of "Get" functions of GATTable.

All of these functions return a `GATResult`. The `GATResult`, covered in Appendix C, indicates their completion status.

F.5 Obtaining a Table's Size

To obtain the number of key/value mapping in a `GATTable` instance, its so called *size*, one uses the following function

```
GATuint32 GATTable_Size(GATTable_const table)
```

The first argument to this function is the `GATTable` instance one wishes to obtain the size of. This function returns a `GATuint32` which is the size of the passed `GATTable` instance. If this function encounters an error of any sort in computing the size of the passed `GATTable` instance, it returns the value `(GATuint32) (-1)`.

F.6 Obtaining All Keys

To obtain all of the various keys contained within a given `GATTable` instance one uses the following function

```
void **GATTable_GetKeys(GATTable_const table)
```

This function takes as its first argument the `GATTable` instance from which the keys are to be obtained. This function returns a `NULL` terminated array of keys contained within this passed `GATTable` instance.

As a convenience there also exists a function which can be used to free all the keys allocated by the above call. It is

```
void GATTable_ReleaseKeys(GATTable_const table, void ***keys)
```

The first argument is the `GATTable` instance from which these keys are culled. The second argument is a pointer to the array of keys. Upon completion this function frees all the keys allocated by the previous call to the "GetKeys" function.

G Appendix: GATString

Instances of the class `GATString` represent character strings, such as "The quick bown fox jumped over the lazy dog."

The C programming language uses an set of rules, an *encoding*, which maps between an array of bytes and a character string. The set of rules employed by the C programming language is called ASCII. ASCII, however, is limited. The range of the ASCII map is extremely limited. For example, the ASCII map does not contain the German character ö in its range. This problem only gets worse for non-Latinate languages such as Madarin Chinese or Arabic.

The class `GATString` class remedies this situation buy allowing for many encodings so that the majority of the world's character strings can be represented as byte arrays. So, for example, the class `GATString` can be used to represent a Mararin Chinese character string, something which was impossible using standard C strings. In addition, the class `GATString` allows for translations between the various encodings. So, for example, one can translate a character string encoded using ASCII in to a character string encoded using UTF-32 say.

In summary, the class `GATString` allows for a platform independent representation almost arbitrary character strings, the majority of which were not possible using an ASCII encoding. In particular, this class supports the following character encodings, organized according to the grouping with which they are associated

- **European languages** ∼ ASCII, ISO-8859-{1, 2, 3, 4, 5, 7, 9, 10, 13, 14, 15, 16}, KOI8-R, KOI8-U, KOI8-RU, CP{1250, 1251, 1252, 1253, 1254, 1257}, CP{850, 866}, Mac{Roman, CentralEurope, Iceland, Croatian, Romania}, Mac{Cyrillic, Ukraine, Greek, Turkish}, Macintosh
- **Semitic languages** ∼ ISO-8859-{6, 8}, CP{1255, 1256}, CP862, Mac{Hebrew, Arabic}
- **Japanese** ∼ EUC-JP, SHIFT_JIS, CP932, ISO-2022-JP, ISO-2022-JP-2, ISO-2022-JP-1
- **Chinese** ∼ EUC-CN, HZ, GBK, GB18030, EUC-TW, BIG5, CP950, BIG5-HKSCS, ISO-2022-CN, ISO-2022-CN-EXT
- **Korean** ∼ EUC-KR, CP949, ISO-2022-KR, JOHAB
- **Armenian** ∼ ARMSCII-8
- **Georgian** ∼ Georgian-Academy, Georgian-PS

- **Tajik** ~ KOI8-T
- **Thai** ~ TIS-620, CP874, MacThai
- **Laotian** ~ MuleLao-1, CP1133
- **Vietnamese** - VISCII, TCVN, CP1258
- **Platform specifics** ~ HP-ROMAN8, NEXTSTEP
- **Full Unicode** ~ UTF-8, UCS-2, UCS-2BE, UCS-2LE, UCS-4, UCS-4BE, UCS-4LE, UTF-16, UTF-16BE, UTF-16LE, UTF-32, UTF-32BE, UTF-32LE, UTF-7, C99, JAVA

G.1 Creating and Destroying String Instances

To create an instance of the class `GATString` one employs the function

```
GATString GATString_Create(const char *byteArray,
                           GATuint32 lengthInBytes,
                           const char *encoding)
```

The first argument to this function is a pointer to an array of bytes containing the encoded string the instance is to represent. The next argument is a `GATuint32` indicating the length in bytes of the first argument. The final argument to this function is a standard C string which is the encoding os the first argument. The function returns a `GATString` corresponding to the passed information or `NULL` upon error.

To destroy a so created `GATString` instance one uses the following function

```
void GATString_Destroy(GATString *string)
```

G.2 Examining a String's Properties

After creating or obtaining a `GATString` instance through other means one can examine various properties of this `GATString` instance. In particular one can obtain the length in bytes of the encoded version of this `GATString` instance through use of the function

```
GATuint32 GATString_GetLengthInBytes(GATString_const string)
```

The first argument is the `GATString` instance to examine. The function returns a `GATuint32` which upon success is the length of the encoded `GATString` instance in bytes; upon failure it is `(GATuint32) (-1)`. To obtain the encoded version of a `GATString` instance one uses the function

```
const char * GATString_GetBuffer(GATString_const string)
```

The first argument is the `GATString` instance to examine. This function returns the encoded version of the passed `GATString` instance. One can also obtain the encoding of a given `GATString` instance through use of the function

```
const char * GATString_GetEncoding(GATString_const string)
```

The first argument is the `GATString` instance to examine. This function returns a standard C string containing the encoding of the passed `GATString` instance.

G.3 Translating a String to a New Encoding

One can translate a given `GATString` instance to a new encoding using the function

```
GATResult GATString_Translate( GATString_const string,
                               const char *newEncoding,
                               GATString *result)
```

The first argument is the `GATString` instance to transcode. The next argument is a standard C string indicating the target encoding. The final argument is a pointer to the transcoded `GATString` instance. This function returns a `GATResult`, covered in Appendix C, which indicates its completion status.

G.4 Comparing two Strings

In analogy to the standard C function `strcmp` there exists a `GATString` function which compares two `GATString` instances. It is the following function

```
GATResult GATString_CompareTo( GATString_const lhs,
                               GATString_const rhs,
                               int *comparison)
```

The first argument is the first `GATString` instance to compare and the second argument is the second `GATString` instance to compare. The final argument is a pointer to a `int` which contains the results of the comparison. The comparison is based on the UTF-32 value of each character in the strings and the indexes into UTF-32 versions of these strings. The character sequence represented by the first `GATString` instance instance is compared "lexicographically" to the character sequence represented by the second `GATString` instance. The result is a negative integer if the first `GATString` instance "lexicographically" precedes the argument second `GATString` instance. The result is a positive integer if the first `GATString` instance "lexicographically" follows the argument second `GATString` instance. This is the definition of lexicographic ordering. If two strings are different, then either they have different characters at some index that is a valid index for both strings, or their lengths are different, or both. If they have different characters at one or more index positions, let k be the smallest such index; then the string whose character at position k has the smaller value, as determined by using the $<$ operator in the UTF-32 encoding, lexicographically precedes the other string. In this case, this function returns the difference of the two character values at position $k$in the two strings. Finally this function returns a `GATResult`, covered in Appendix C, which indicates its completion status.

G.5 Examining String Prefix and Suffix

One can determine if a given `GATString` instance ends with a second `GATString` instance using the function

```
GATResult GATString_EndsWith( GATString_const string,
                              GATString_const query,
                              GATBool *result)
```

This function takes as its first argument the "target" GATString instance. The next argument is the "query" GATString instance. The final argument is a pointer to a GATBool, a type covered in Appendix B, which returns a boolean indicating if the "target" string ends with the "query" string. Finally this function returns a GATResult, covered in Appendix C which indicates its completion status.

One can determine if a given GATString instance starts with a second GATString instance using the function

```
GATResult GATString_StartsWith( GATString_const string,
                                GATString_const query,
                                GATBool *result)
```

This function takes as its first argument the "target" GATString instance. The next argument is the "query" GATString instance. The final argument is a pointer to a GATBool, a type covered in Appendix B, which returns a boolean indicating if the "target" string starts with the "query" string. Finally this function returns a GATResult, covered in Appendix C which indicates its completion status.

G.6 Concatenating Strings

One can concatenate two GATString instances using the function

```
GATResult GATString_Concatenate( GATString_const head,
                                 GATString_const tail,
                                 GATString *result)
```

The first argument is the GATString instance which is to be at the "head" of the resultant GATString instance. The next argument is the GATString instance which is to be at the "tail" of the resultant GATString instance. The final argument is a pointer to a GATString through which the concatenation is returned to the caller. Finally this function returns a GATResult, covered in Appendix C which indicates its completion status.

G.7 Examining Substrings of a String

One can determine the last "index" of a "query" GATString instance in a "target" GATString instance through us of this function

```
GATResult GATString_LastIndexOf( GATString_const target,
                                 GATString_const query,
                                 GATuint32 *queryIndex)
```

This index is based upon UTF-32 encoded versions of the "target" and "query" strings. This first argument is the "target" GATString instance which is to be examined. The second argument is the "query" GATString instance whose index in the "target" GATString instance is to be determined. The final argument is a pointer to a GATuint32 used to return to the caller the index of the "query" GATString instance in the "target" GATString instance. If the "query" GATString instance does not occur in this "target" GATString instance, the value returned, upon no errors being encountered, is `(GATuint32) (-1)`. Finally, this function returns a `GATResult`, covered in Appendix C which indicates its completion status.

One can determine the first "index" of a "query" `GATString` instance in a "target" `GATString` instance through us of this function

```
GATResult GATString_FirstIndexOf( GATString_const target,
                                  GATString_const query,
                                  GATuint32 *queryIndex)
```

This index is based upon UTF-32 encoded versions of the "target" and "query" strings. This first argument is the "target" `GATString` instance which is to be examined. The second argument is the "query" `GATString` instance whose index in the "target" `GATString` instance is to be determined. The final argument is a pointer to a `GATuint32` used to return to the caller the index of the "query" `GATString` instance in the "target" `GATString` instance. If the "query" `GATString` instance does not occur in this "target" `GATString` instance, the value returned, upon no errors being encountered, is `(GATuint32) (-1)`. Finally, this function returns a `GATResult`, covered in Appendix C which indicates its completion status.

G.8 Obtaining Substrings

One can obtain a substring of a given `GATString` instance using the function

```
GATResult GATString_GetSubString( GATString_const target,
                                  GATuint32 start,
                                  GATuint32 end,
                                  GATString *substring)
```

The first argument to this function is the "target" `GATString` instance from which the substring will be extracted. The second argument is a `GATuint32` indicating the index of the first character in the resultant substring. This index is computed using a UTF-32 representation of the "target" `GATString` instance. The next argument `end` dictates the last character in the resultant substring. The last character in the resultant substring is the character at the index `(end - 1)`, again this index is computed using a UTF-32 representation of the "target" `GATString` instance. The next argument to this function is a pointer to a `GATString`. This is used to return to the caller the resultant substring. Finally, this function returns a `GATResult`, covered in Appendix C which indicates its completion status.

H Appendix: Backus-Naur Form

The Backus-Naur form (BNF) (also known as Backus normal form) is a metasyntax used to express context-free grammars: that is, a formal way to describe formal languages[12].

It is widely used as a notation for the grammars of computer programming languages, command sets and communication protocols; most textbooks for programming language theory and/or semantics document BNF. Some variants, for example ABNF, have their own documentation..

It was originally named after John Backus and later (at the suggestion of Donald Knuth) also after Peter Naur, two pioneers in computer science, namely in the art of compiler design, as part of creating the rules for Algol 60.

A BNF specification is a set of derivation rules, written as

```
<symbol> ::= <expression with symbols>
```

where `<symbol>` is a nonterminal, and the expression consists of sequences of symbols and/or sequences separated by '|', indicating a choice, the whole being a possible substitution for the symbol on the left. Symbols that never appear on a left side are *terminals*. Symbols inside brackets [ ] are optional.

Example:
As an example, consider this BNF for a US postal address:

```
<postal-address> ::= <name-part> <street-address> <zip-part>
<personal-part> ::= <name> | <initial> "."
<name-part> ::= <personal-part> <last-name> [<jr-part>] <EOL> |
                <personal-part> <name-part>
<street-address> ::= [<apt>] <house-num> <street-name> <EOL>
 <zip-part> ::= <town-name> "," <state-code> <ZIP-code> <EOL>
```

This translates into English as:

> "A postal-address consists of a name-part, followed by a street-address part, followed by a zip-code part. A personal-part consists of either a first name or an initial followed by a dot. A name-part consists of either: a personal-part followed by a last name followed by an optional "jr-part" (Jr., Sr., or dynastic number) and end-of-line,

[12] This chapter is adapted from the Wikipedia Backus-Naur Form entry.

> or a personal part followed by a name part (this rule illustrates the use of recursion in BNFs, covering the case of people who use multiple first and middle names and/or initials). A street address consists of an optional apartment specifier, followed by a street number, followed by a street name. A zip-part consists of a town-name, followed by a comma, followed by a state code, followed by a ZIP-code followed by an end-of-line."

Note that many things (such as the format of a personal-part, apartment specifier, or ZIP-code) are left unspecified here. If necessary, they may be described using additional BNF rules, or left as abstractions if irrelevant for the purpose at hand.

Variants

There are many variants and extensions of BNF, possibly containing some or all of the POSIX regular expression wild cards, see Appendix E. The Extended Backus-Naur form (EBNF) is a common one. In fact the example above isn't the pure form invented for the ALGOL 60 report. "[ ]" was introduced a few years later in IBM's PL/I definition but is now universally recognised. ABNF is another extension.

I Appendix: POSIX Paths

A POSIX path is a string which is used to identify a directory or file in an operating system independent manner. Unlike a URL it does not specify the protocol which is to be used to access the directory or file, it is usually the case that they are accessed through the operating specific mechanisms. The BNF grammar for a POSIX path is as follows, for the reader unfamiliar with BNF grammars refer to Appendix H:

```
path ::= [root] [relative-path]
root ::= [root-directory]
root-directory ::= "/"
relative-path ::= path-element { "/" path-element } ["/"]
path-element ::= name | parent-directory | directory-placeholder
name ::= char { char }
directory-placeholder ::= "."
parent-directory ::= ".."
```

This grammar is supplemented with the restriction that `char` may not be '/' or '\0' and an empty path is also valid. The optional trailing "/" in a relative-path is allowed as a notational convenience. It has no semantic meaning and is simply discarded.

J Appendix: GATList

An instance of the class GATList is used to group an ordered set of elements. The story of the class GATList, however, is not that simple. The class GATList can, properly, be thought of as a "template class." A template class is a class that creates other classes. For example, if one needs a class `GATList_GATFile` which is used to group an ordered set of `GATFile` instances, then the template class GATList can be used to construct the class `GATList_GATFile`. More formally, one would indicate the fact that the template class GATList can create various other classes, e.g. classes of the form `GATList_GATFile`, by writing the template class GATList in the following manner `GATList_<T>`. The addition of the `_<T>` to the end of GATList indicates that GATList is a template class and that the template `<T>` is bound to a certain class, e.g. `GATFile`, to create a new class of the form `GATList_GATFile` for example.

Beyond grouping an ordered set of elements, the template `GATList_<T>` class also allows for one to iterate – forward or backward – through these elements, get an element from the list, remove an element from the list, find the size of the list, and a set of various other functionalities. Let us begin our study of the functionalities of the template class `GATList_<T>`.

J.1 Creating and Destroying List Instances

As with the majority of GAT classes, creating and deleting instances is rather simple, and a good place to begin with when learning the in's and out's of a new GAT class. The template class `GATList_<T>` is no exception. To create an instance of the class `GATList_<T>` one use the function

```
GATList_<T> GATList_<T>_Create( void );
```

This function will return a instance of type `GATList_<T>` upon success. Upon failure it will return a `NULL` value. So, for example, if one is dealing with a list in which the template is bound to the class `GATFile`, then one would create the corresponding list with the following function

```
GATList_ GATFile GATList_GATFile_Create( void );
```

Destroying a list is also simple. One calls the function

```
void GATList_<T>_Destroy(GATList_<T> *list);
```

The first argument to this function is a pointer to the list one wishes to destroy. Upon successful completion, this function releases and resources held by the passed list. This includes any resources held by the contained list elements. For example, if one is dealing with a list in which the template is bound to the class `GATFile`, then one would destroy the list with the following function call

```
void GATList_GATFile_Destroy(GATList_GATFile *list);
```

Again, this function upon successful completion will release all resources held by the passed list and all resource held by the elements contained within the passed list, i.e. it will call the function `GATFile_Destroy` on each of the contained `GATFile` instances.

J.2 Examining a List's Properties

As all instances of the template class `GATList_<T>` are subclasses of the superclass `GATObject` they inherit all of the standard `GATObject` appendages such as an "Equals" function, a "Clone" function, a "GetType' function, and on and on. So, one can use these to examine the properties of a `GATList_<T>` instance. In addition to these standard functions there exists a function which obtains the number of elements in a given list. It is as follows

```
size_t GATList_<T>_Size(GATList_<T>_const list);
```

It takes as its first argument the list which one wishes to examine. It returns a value of type `size_t` indicating the number of elements the passed list contains. So, for example, in the case of a list in which the template is bound to `GATFile` this function would have the form

```
size_t GATList_GATFile_Size( GATList_GATFile_const list);
```

J.3 Obtaining Iterators

An iterator is an opaque type, not a subclass of `GATObject`, which is used to specify an element in a `GATList_<T>`. For example an iterator may specify the first element in a list, the second element in a list, or any other element in a list. One can think of an iterator as somewhat akin to a pointer pointing to a list element. It is through the use of these iterators that one is able to refer to differing elements in a list. Now lets see how to obtain various iterators.

J.3.1 Obtaining the "Begin" Iterator

The firs iterator we are going to learn how to obtain is the "begin" iterator which points, you guessed it, to the beginning of the list with which it is associated. This iterator is obtained through a call to the function

```
GATList_<T>_Iterator GATList_<T>_Begin(GATList_<T>_const list);
```

The first and only argument to this function is an instance of type `GATList_<T>`. This is the list for which one wishes to obtain the "begin" iterator. (Note, that each iterator is associated with one, and only one, list. It is an error to use an iterator with a list with which it is not associated.) This function, upon success, returns a `GATList_<T>_Iterator` which "points" to the beginning element in the passed list. Upon failure this function returns `NULL`.

As a concrete example of this function in action consider obtaining the begin iterator for a list in which the template is bound to the type `GATFile`. This would necessitate a call to the function

```
GATList_GATFile_Iterator GATList_GATFile_Begin(GATList_GATFile_const list);
```

Let us create a code snippet which wraps this function and instead of returning a `NULL GATList_GATFile_Iterator` upon failure returns a `GATResult` indicating the completion status of the function. This function could be written as follows

```
GATResult GATList_GATFile_Begin_Wrapper( GATList_GATFile_const list,
                                         GATList_GATFile_Iterator *iterator )
{
  GATResult result;
  GATList_GATFile_Iterator swapIterator;

  result = GAT_MEMORYFAILURE;

  swapIterator = GATList_GATFile_Begin( list );

  if( NULL != swapIterator )
  {
    *iterator = swapIterator;
    result = GAT_SUCCESS;
  }

  return result;
}
```

J.3.2 Obtaining the "End" Iterator

As we've now seen how to obtain the begin iterator, one might guess that the next function we'd cover would obtain the "end" iterator. This is indeed the case. The "end" iterator can be obtained through a call to the function

```
GATList_<T>_Iterator GATList_<T>_End(GATList_<T>_const list);
```

The first argument to this function is the `GATList_<T>` instance for which one wishes to obtain the end iterator. The function, upon success, returns a `GATList_<T>_Iterator` which points to the end element in the passed list.

J.3.3 Obtaining the "Next" Iterator

After obtaining an iterator – be it the first, last, or other iterator – it is often the case the one wishes to obtain an iterator pointing to the next element in the list. This is done through a call to the function

```
GATList_<T>_Iterator GATList_<T>_Next(GATList_<T>_Iterator_const iterator);
```

This first argument to this function is the iterator, of type `GATList_<T>_Iterator`, for which one wishes to find the "next" iterator. Upon success this function returns a `GATList_<T>_Iterator` which points to the list element after the list element pointed to by the passed `GATList_<T>_Iterator`. Upon failure it returns `NULL`.

J.3.4 Obtaining the "Previous" Iterator

Obtaining the "previous" iterator is done through a call to the function

```
GATList_<T>_Iterator GATList_<T>_Previous(GATList_<T>_Iterator_const iterator);
```

This first argument to this function is the iterator, of type `GATList_<T>_Iterator`, for which one wishes to find the "previous" iterator. Upon success this function returns a `GATList_<T>_Iterator` which points to the list element before the list element pointed to by the passed `GATList_<T>_Iterator`. Upon failure it returns `NULL`.

J.4 Obtaining List Elements

We've now covered how to create and destroy lists, examine a lists, and how to obtain iterators over lists. Next we'll cover how to obtain a list element pointed to by a particular iterator. This is done through a call to the function

```
<T> * GATList_<T>_Get(GATList_<T>_Iterator_const iterator);
```

This function takes as its first argument a `GATList_<T>_Iterator` pointing to the list element one wishes to obtain. This function returns a pointer to a `<T>`. Upon success this pointer points to the list element indicated by the passed `GATList_<T>_Iterator`. Upon failure this pointer is `NULL`.

J.5 Adding and Removing List Elements

In addition to examining a pre-populated list one can also add or remove elements from a list.

J.5.1 Adding List Elements

To add an element to a list one uses the function

```
GATList_<T>_Iterator GATList_<T>_Insert(GATList_<T> list,
                                        GATList_<T>_Iterator iterator,
                                        <T> element);
```

The first argument to this function is the `GATList_<T>` into which one wishes to insert a new element. The function's second argument is a `GATList_<T>_Iterator` which is used to indicate where to insert this new list element. Upon success the new list element will be inserted directly before the list element pointed to by this iterator. The next argument to this function is a `<T>`, the new list element to be added. Finally, this function returns a `GATList_<T>_Iterator` which upon success is a iterator pointing to the newly inserted element. Upon failure this function returns `NULL`.

J.5.2 Removing List Elements

To remove an element to a list one uses the function

```
GATList_<T>_Iterator GATList_<T>_Erase(GATList_<T> list,
                                       GATList_<T>_Iterator iterator);
```

The first argument to this function is the `GATList_<T>` from which one wishes to remove an element. The function's second argument is a `GATList_<T>_Iterator` which is used to indicate which list element to remove. Upon success the list element pointed to by the passed iterator will be removed. Finally, this function returns a `GATList_<T>_Iterator` which upon success is a iterator pointing to the element following the removed element. Upon failure this function returns `NULL`.

J.6 Splicing Lists

In addition to all the functionality presented, one can also splice lists together. In other words one can remove a sequence of list elements from a source list and insert this sequence into a destination list. This functionality can be accomplished with the primitives we have presented; however, the splice functionality serves to wrap up this functionality into a nice useful package.

The splice functionality is accomplished through a call to the function

```
GATList_<T>_Iterator GATList_<T>_Splice(
  GATList_<T> dest, GATList_<T>_Iterator here,
  GATList_<T> src, GATList_<T>_Iterator first, GATList_<T>_Iterator last,
  size_t count)
```

The first argument is the destination list, the `GATList_<T>` in to which elements will be placed. The second argument is a `GATList_<T>_Iterator` indicating where the new elements should be placed. The new elements will be placed directly before the list element referenced by this iterator!!! The function's next argument is the source list, the `GATList_<T>` from which elements will be taken. The next argument is an iterator which should point to the first element in the source list to transfer. The following argument is an iterator which points to the last element in the source list which is to be transfered to the destination list. The next argument is the number of elements to transfer, this parameter may be set to 0 in which case the number of elements is computed. This final argument simply exists as an optimization. Finally this function, upon success, returns an iterator which points to the first element beyond the newly inserted sequence. Upon failure this function returns `NULL`.