1/23/98 11:36 AM

Annual Technical Summary Application Gateway System

NSF Grant Number: NCR-9302522

January 96 to December 96

Corporation for National Research Inititives

1895 Preston White Drive, Suite 100

Reston, VA 20191

Introduction

This report describes progress on the Application Gateway System (AGS) during the period beginning January 1, 1996 and ending December 31, 1996. The Application Gateway System is a collection of software components which provides a robust substrate that operates a collection of servers using distributed load-balancing techniques and secure remote management. The AGS is composed of routers, an AGS Management System and a collection of computers that host the application servers. Each host is called an Application Gateway (AG). An AGS may be comprised of a collection heterogeneous computers (AGs) and operating systems of varying capabilities.

Several applications and core technologies for the AGS system were developed and demonstrated during this reporting period:

1. Two implementations of the rate.d system were developed to provide load-balanced high availability access to a single service. The first demonstration used a simple "delay service" to simulate client requests of variable resource demands, while the second simulated multiple service vending by the AGS.
2. A set of GUI tools to monitor and control a rate.d balanced AGS through the use of the ILU distributed object system. The rate.d system and monitor tools were demonstrated to D. Lindberg (NLM), B. Leiner and R. Larsen (DARPA), and M. Nelson (OSTP)
3. MEDLINE data was integrated into the Handle System. A sub-collection (approximately 500,000 records) of the MEDLINE database was installed on the CNRI Handle System, and in a separate repository for test and evaluation purposes. This demonstrated the ability to specify and load abritrary data types (such as the Information Sources Map data) into the Handle System. This enabled the demonstration of access to the MEDLINE data through a variety of mechanisms. The first access mechanism involved the use of Elhill for searching, via the Python language Expect modules, resulting in a list of Elhill unique identifiers. These identifiers were used to retrieve MEDLINE records directly from Elhill, from the repository, and directly from the Handle System. In the case of direct access of MEDLINE records from Elhill, these records were used to populate the repository and the Handle System, so that a mechanism for migration of MEDLINE records from Elhill to alternative storage facilities could be demonstrated. The Elhill record migration to the repository and the Handle System allowed for the demonstration of search and retrieval tools involving the extensible Internet browser Grail, Knowbot Programs and the Knowbot Service Station.
4. The GrailMED application was demonstrated, showing a graphical, Web-based mechanism for performing search and retrievals on MEDLINE data using the various mechanisms described herein. GrailMED was further enhanced to support the creation and tracking of Knowbot Programs to perform searches, thus demonstrating the ability to move processing closer to the data. A GrailMED applet was also demonstrated that aided users in creating and configuring Knowbot Programs specifically for searching the Elhill databases using the mechanisms described in the GrailMED section of this report. A scripting language called Python that is easy for nonprogrammers to understand is used to customize Knowbot Programs thus allowing nonprogrammers to customize existing Knowbot Programs and to create their own.

Project Goals

The goal of the AGS project is to provide a scalable system from which network services such as HTTP, FTP, query engines, etc., can be vended. The goal can be illustrated by two hypothetical scenarios:

The first scenario illustrates the use of the AGS to off load connection time from a central database server to a collection of front-end machines. Consider the current Elhill environment, clients make connections to Elhill, perform searches, and then may spend a considerable amount of time reading the results of these searches. While clients are reading the results, although primarily idle, the connections are still in place, consuming valuable resources. By migrating Elhill access to the AGS, idle time can be pushed onto the individual AGs, which have much greater capacity. This reduces the burden on Elhill itself so that its resources are consumed only by the amount of work necessary to perform the search and transmit the results to an AG. This should increases the overall throughput of Elhill.

For the second scenario consider a moderate to large sized software developer named SD with tens of millions or more copies of its products in daily use. SD conducts its business over the Internet. It posts on its Web pages helpful hints, answers to frequently asked questions, patches to fix bugs, and it sells and delivers its software over the Internet. All the functions are handled by HTTP (Web pages), except delivery of software which is accomplished by FTP. While access to a Web page can be load leveled by replicating it and accessing one of the replicated pages via a round-robin DNS lookup, one must manually choose an FTP site from a list of them for the software delivery process. When SD releases a new version of its popular software, Web page access demand drops and FTP demand explodes as people rush to download the latest version of the software.

The AGS allows SD to handle this situation gracefully by deploying its server resources between HTTP and FTP and automatically providing access to the least loaded FTP server instead of forcing the user to choose (from a list) a server that may already be loaded to capacity. Thus the AGS provides a seamless and uniform access mechanism to its client base, i.e., for each service, access is provided through a virtual IP address but clients are served from a scalable number of actual servers deployed according to the load. The load for each service is balanced among all the servers by a system called rate.d which uses the actual capability of the servers at the time of the request for service. Thus an AGS allows SD to manage its server resources in a more effective manner then is possible today. In addition, SD provides for its customers a smoother and easier to use set of services each via a single virtual net address.

Rate.d Implementation

The AGS uses a system called "rate.d" designed for the AGS to perform load balancing operations across the participating AGs. Its design and protocol were discussed extensively in the 1995 report. This report discusses implementation work performed in 1996.

The rate.d algorithm is implemented as a daemon process on each AG. Each AG communicates its status to the others whenever its status changes. Using the status information previously obtained, each AG's rate.d daemon process computes which AG is the best able to service new request. There are three interesting cases: 1. Each AG identifies the same unique AG as the best one to provide the service, that AG answers, and the request is satisfied by that AG. 2. Two or more AGs determine that they are the best candidate to service the request and each will respond. The first responding AG will get the job and the others will be ignored. 3. AG A will determine that AG B is the best AG to provide the service and vice versa. In this case a time-out will occur without a response and the presumed second best AG will respond with the possibility of reverting to the situation of case 2.

Two implementations of the rate.d daemon were developed to test the algorithm's ability to produce the expected results. Both implementations were written in Python and share a large amount of code. The first implementation supports only services which operate using a single port for all client contact. This implementation limits an AGS to hosting a single service, i.e., FTP, HTTP, etc. The test service used a single UDP socket to receive messages. Although not tested, the same daemon would be able to respond to multicast requests as well. The second implementation supported multiple services on an AG. The

Inter-Language Unification (ILU) distributed object system from Xerox PARC was used to implement this service. In particular, the implementation added the ability to communicate with the service to create new application interfaces and return the appropriate object binding information needed to allow service clients to connect with the service. This allowed experimentation with placing multiple services on a single AG.

A process was implemented that monitored activity for all the rate.d daemons participating in an AGS. The monitor process allows short-lived programs to collect and report the status of the AGS to the user or administrator. Since the information is made available by a well-defined interface, different programs could be written which made use of the information in different ways. More importantly, using a distributed object system facilitates notification of the monitor clients when information is changed. This functionality is used in the demonstration applets discussed below. The published interface to the monitoring process provides access to control features of the rate.d daemons as well.

To support the demonstration of rate.d capabilities, a simple service and a simple client were implemented. The services implemented a simple "delay service" which sent the client a short message after a specified delay. This test service was chosen to emulate the period of time before the completion of a request, independent of the speed of the system hosting the service. This attribute is useful when visually displaying activity of an AGS. The clients for the delay services issued requests for service repeatedly based on command line parameters. Future work will concentrate on more realistic jobs with real workloads attached, possibly including FTP, HTTP, and more computationally intense services.

System activity is displayed and controlled using applets running in an Internet Web browser. The applets work as follows: They create ILU objects that allow the monitor process to call them whenever AGS state changes and the results update the information displayed in the browser. The browser was modified to support integration with ILU. Two applets were written which interface to the rate.d monitor process: one displays the status of a single rate.d daemon, and the other controls the number of jobs accepted by a single application gateway at any one time. The approach of using small applets which perform more discrete amounts of work was taken to allow the applets to be used as building blocks in larger applications. Such applications could control user access to HTML pages using the applets via existing HTTP access control mechanisms. Future versions of the AGS will contain access control to remove this dependency; this will become an attribute of the AGS management system.

Design Considerations Related to Supporting Multiple Services

The goal of the AGS is to support multiple services concurrently on an AGS where each AG may run some or all services. This is illustrated in Figure 1. (Note, the figure indicates the design of the AGS and is not intended to imply it is implemented and operational.) The AGS may be configured to provide optimum service for the collection of services it is vending. For instance, at the time of a software release, an AGS may be configured to provide maximum throughput for FTP service dispensing the software at the expense of HTTP services that are unrelated to the software release. After a few days, when the demand for the newly released software diminishes, the AGS may be reconfigured for its normal operation. Configuration of the AGS will be accomplished from the management workstation and can be done while the AGS is in operation.

The initial implementation of rate.d was solely for UDP- and TCP-based services and supported only a single service; all systems in the AGS ran the same service. The current implementation is more flexible, allowing multiple services to run on each system. It has the limitation that each AG must run all the services offered by the AGS. While sufficient for debugging purposes, the more general case is where not all services may be provided by each host in the AGS, and each rate.d daemon know what services are available at each host. This will be available late in 1997.

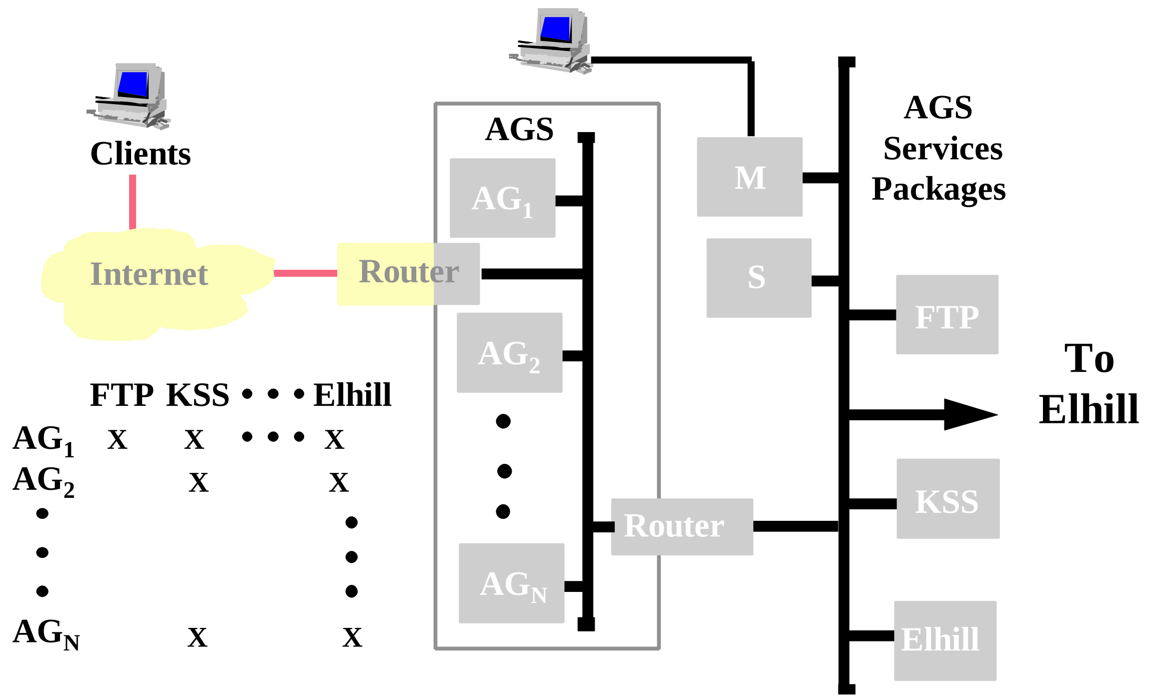


Figure 1. Application Gateway System -- Three services are vended on the AGS: FTP, KSS, and a set of front-ends to Elhill. FTP runs on one AG, KSS on three and the Elhill search on all. The multi-shaded router presents a single virtual address to Internet clients for each service it is vending and translates that virtual address to a real local address of an AG, running the service, that is best able to handle the request. It also acts as a firewall protecting the services and the AGs from all except legitimate requests for service.

The services that the AGS runs at any time and on which AGs they run will be controlled by a AGS Management Station shown in Figure 1 as the box labeled “M”. It is used to monitor and control operation of the AGS. The design of it and the management controls that run on each AG is in progress. A new protocol is being defined, the AGS Control Protocol, that will allow the Management System secure access to the AGS daemons for configuring the operations desired on each host participating in the AGS. A set of commands are being designed that allow authorized administrators to control the AGS via command-line options or via a graphical user interface which provide a mechanism for on-the-fly tuning of the service mix. The box labeled “S” in Figure 1 ensures that access is achieved by only authorized AGS administrators. Our intent is to use available security systems, perhaps Kerberos and/or a token card system.

A GUI Interface and Performance Monitoring and Control Applets

In order to demonstrate the functionality of rate.d, some means of displaying the activity of the individuals nodes of an AGS as well the ability to control the operation of the rate.d daemons is required. To limit the amount of user-interface coding needed and to support flexible mechanisms to construct the demonstration from, each display element was constructed as an applet with a separate connection to the rate.d monitor process. The ILU Interface Specification Language (ISL) is used to define an interface specification to support communication between the monitor process and the applets. Details are discussed in Appendix A.

Three separate applets were constructed, two of which relate specifically to rate.d and one which was used to improve the display. Figure 2. shows the display of a monitor run by a browser whose active part is controlled by three applets. The first applet displays the status of a single rate.d daemon as a "progress bar": a bar of a fixed width is displayed which shows two overlapping colored-bar indicators. One bar indicates the number of jobs currently active for the host and the second indicates the number of jobs allowed for that host. The area of the second bar remaining after overlap indicates the remaining system capacity which may be allocated via rate.d. This applet takes nine optional arguments including the hostname of the rate.d process to monitor and addressing information for monitor process. Some parameters are used to control display attributes such as the colors used for the various bars.

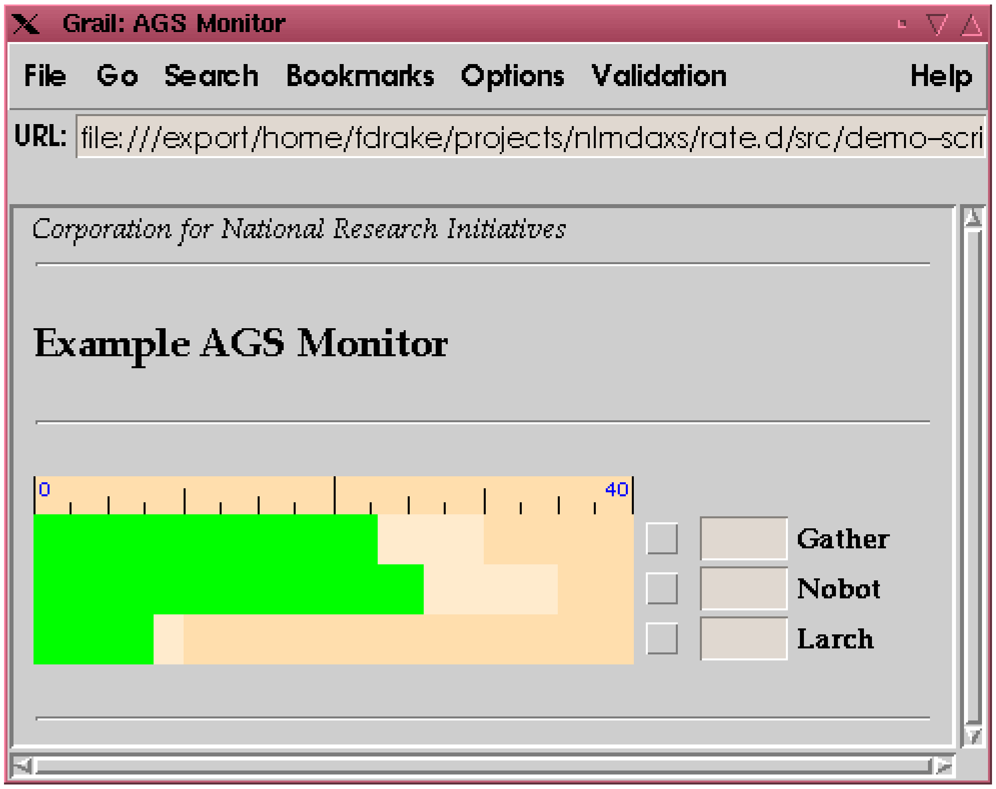

Grail: AGS Monitor
File
Go
Search
Bookmarks
Options
Validation
Help
URL:
file:///export/home/fdrake/projects/nlmdaxs/rate.d/src/demo-scri
Corporation for National Research Initiatives
Example AGS Monitor
0
40
Gather
Nobot
Larch

Figure 2. A Browser running the Monitor and Controller Applets

The second applet is used to control the rate.d daemon at a single host and has two input fields. The first is a checkbox which can be toggled with a mouse click to enable or disable administrative limiting of resources dedicated to the jobs processed by rate.d. The second field is a type-in box which allows the entry of a number; this is used to set the administrative limitation on resource allocation. Entering a number in this box and pressing <Enter> will cause the limitation to be set and the checkbox indicator to be marked. Two parameters must be supplied to the applet to allow it to correctly address a host through the monitor process.

The third applet generates some ruler-like markings for the display area allocated to. This is used to augment the appearance of HTML pages which use these applets.

Integrating MEDLINE data into the Handle System

A handle is a name used to identify Internet resources. The handle and its associated information is called the handle record. The Handle System, developed with DARPA support, is a distributed computer system that stores handle records and provides the resolution of a Handle into information that is used to locate and access the identified resource.

The Local Handle Server system allows globally registered authorities to establish handle records which are maintained on a machine within a local administrative domain instead of on the global handle server. Local Handle Servers also provide an ideal platform for organizations to experiment with the CNRI handle protocol.

During this period the Handle System was used to experiment with loading and accessing Information Sources Map data. This data is used as a master index of known information sources on a particular subject. A Local Handle Server system was installed at NLM to allow NLM researchers to experiment with this data.

To enable the loading of NLM MEDLINE records directly into the handle system, a new feature of the CNRI handle system was employed: user defined record types. A user defined record type allows the handle system to recognize and specify data types not configured by default in the handle system software. In the case of NLM, we defined a user defined data type of "NLM" which was a record type containing a single MEDLINE record. This is similar to specifying schema information in a database management system with one important exception: the handle system does not specify or care about the contents of these new record types. It simply identifies them uniquely.

Loading full MEDLINE records as a user defined record type in the CNRI handle system proved to operate quite well, but using the handle system as a repository is not its intended use. Rather then housing the entire NLM record, the handle system typically provides identifiers associated with access data to the desired information. A URL handle data type is an example. NLM information sources map records are another. By providing a live example of NLM data defined as a user defined data type in the handle system, a proof-of-concept was established for populating the handle system with Information sources map data as well. CNRI did a proof of concept and NLM (R. P. C. Rodgers) suggested that they undertake further experiments with this capability.

Large Handle Records for MEDLINE

Being primarily UDP based, most handle records are returned in one data packet, e.g. when the data is type URL. When the data returned by a request is of user defined type NLM, the average data returned is 6-8 UDP packets. Support for continuation packets was added to the Python handle resolution client library during this period along with support for proper ordering of UDP packets received out of order.

During this period, CNRI obtained a tape from NLM that contained approximately 500,000 MEDLINE records. These MEDLINE records were loaded into two services: a Repository; and the Handle System itself. The repository made use of the UNIX file system. For data storage, a simple hash mechanism via GNU dbm files was employed. Access to records was implemented using the HTTP protocol. A custom HTTP server daemon was constructed that takes URLs with query strings of the following form:

```
http://gather.cnri.reston.va.us:8000/?database=medline&command=lookup&key=c10101
```

and returns the MEDLINE record, formatted as HTML, that corresponds to a record with an Elhill unique identifier of c10101.

For the purpose of experimenting with the MEDLINE data, handles were created for all 500,000 MEDLINE records. These handles were of the form:

```
nlm.hdl_test/c10101
```

Access to these handles through an Internet Browser was implemented via the handle protocol. Resolving the above handle in the browser would be done with the following handle in URL syntax:

```
hdl:nlm.hdl_test/c10101
```

In this example, hdl tells the browser how to treat what follows, "nlm.hdl_test" is the naming authority. What follows the slash is the string which must be unique within naming authority. In the case of the handles corresponding to MEDLINE records the string is the unique identifier corresponding to the MEDLINE record from the record data itself.

The MEDLINE record handles were added to the handle system with two data types: URL and NLM . The URL data type data was the URL that could be processed against the repository for retrieval of the record data. Resolving a handle to it's URL data type actually uses the handle system as a pure resolution system. The location independent name of the record is resolved into a location dependent pointer to the record. Conversely, the NLM data type actually contains the data directly in the handle record.

When resolving a handle into its handle record, the default behavior is to return the whole handle record. Because the handle record may contain more then one data type certain applications may specify a subset of the types to return. In order to facilitate this selection, an option field was added to the handle string. The unique identifier string is followed by a semicolon followed by a key=value pairing. To specify NLM data from the handle server for the example MEDLINE records, the Handle would be:

```
hdl:nlm.nlm_test/c10101;type=nlm
```

Once the MEDLINE records were loaded into the repository and the Local Handle server, a method was developed for retrieving and viewing the records. MEDLINE handle anchors were embedded in HTML pages so one could click on them to retrieve the data. In addition the data could be pulled nearly transparently from the repository, using the handle system as a pure resolution service, or directly from the handle system itself. The only detectable difference was if a handle resolved to an URL, that URL string was left in the URL Entry field in the browser. If the data was obtained directly from the handle system the handle string remained in the URL entry field. This is illustrated in Figure 8 in Appendix B.

GrailMED

Searching vs. Retrieval

Neither the repository nor the handle system has searching capabilities. In order to retrieve a record from either place, its handle (or at least its URL in the case of the repository) must be known. In order to facilitate searching the MEDLINE data while still allowing storage and retrieval to come from either the repository or the handle server, CNRI developed a Grail applet that allows a user to search NLM's Elhill mainframe computers using an ordinary MEDLARS account. This applet, called GrailMED , uses a Telnet connection over the Internet to medlars.nlm.nih.gov and software called Expect to automate an Elhill session. The applet uses the Tk toolkit from Sun Microsystems. The applet is loaded by Grail upon reading the appropriate HTML mark-up. A sample applet GrailMED dialog is shown in Figure 7 in Appendix B.

The use of a Knowbot Program is an alternative method to carry out database searches. It is based on the premise of moving the search function closer to the database repository. This is a more efficient use of network bandwidth when compared to moving large amounts of data across the network to be discarded by a filter on the client.

Knowbot Programs are platform independent and may connect to arbitrary services, such as databases through the use of extensions. Extensions (or plugins), are implemented in a separate process than that of the Knowbot Program, but they are tightly coupled to it. For the purposes of this effort, a plugin was developed that allows Knowbot Programs to communicate with Elhill. The interface for MEDLINE searches, full record retrievals, and short record retrievals is described in Appendix A in the form of an ILU Interface Specification Language (ISL) file.

The Knowbot system software includes a number of tools that help one develop, debug, use, and demonstrate Knowbot Programs. One of these tools, called the visualizer, displays instances of Knowbot Service Stations along with their plugins, as well as any Knowbot Programs that happen to be present. If a Knowbot Program migrates to a different host, the visualizer shows this by moving the Knowbot Program's icon to the

new location as well as drawing an animation arrow depicting the motion. Figure 3 shows a visualizer instance with two Knowbot Service Stations, one of which has an Elhill plugin.

A Knowbot Service Station (KSS) that creates a Knowbot Program acts as a home base for that Knowbot Program. Specifically, the KSS is the recipient of status information regarding Knowbot Program events such as migration or termination. The KSS can be configured to also capture the standard input and output streams of Knowbot Programs as well. In this way, a Knowbot Program can ask its home base for further instructions upon encountering a stopping point that was programmed into the Knowbot Program prior to its execution.

Tracking Knowbot Program Progress with the Visualizer

The search conducted by a Knowbot Program can be tracked with the visualizer. Figure 4 shows an animation depicting the movement of a Knowbot Program between its creating KSS and a KSS with an Elhill plugin. This figure is meant to show a short chronology.

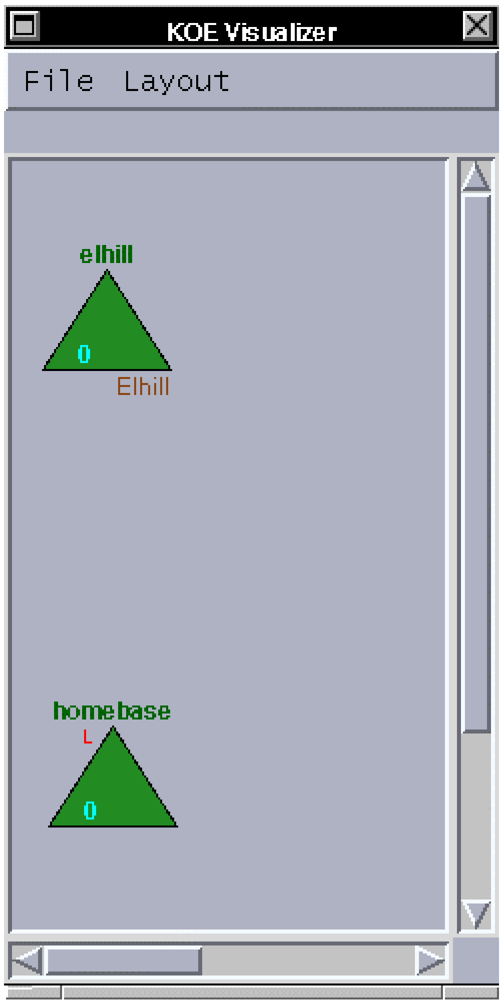


Figure 3. A Knowbot Program Visualizer

The viewing order should be: top-left, top-right, bottom-left, bottom-right. In the top-left panel Knowbot Program GrailMED 123 is created. In the top-right panel the Knowbot Program has migrated to the KSS with the Elhill plugin where the search is executed. In the bottom-left panel the Knowbot Program has completed the search, cloned itself, and terminated its operation. In the bottom-right the terminated Knowbot Program is sent back to its creating KSS leaving a clone of itself monitoring the Elhill repository for new information that fit its search criteria.

ILU Integration with Grail

To support applets which use a distributed object system to communicate with other objects on the network, Grail needs to allow applets to use dynamically loaded extension modules (written in C) associated with the ILU distributed object system. For many applications, this is sufficient, allowing applets to communicate with remote objects but not publish objects. This is not sufficient for applets which wish to publish objects using the distributed system. This problem is created by the need for the object request broker to be integrated with the main Tk event loop. This integration is required to support the correct handling of incoming object invocations across the network.

In Grail, this requirement is supported by allowing the user to set a preference controlling whether the distributed object support should be enabled. When Grail starts, an integrated main loop is used if the preference is enabled. Different distributed object systems may impose different integration requirements, Grail currently only supports the Inter Language Unification (ILU) system from Xerox PARC since that is already in use for other projects at CNRI. In particular, ILU was used in implementing the rate.d control and monitoring process to which the applets built for the rate.d demonstration were required to interface. Other systems based on, for instance, the Object Management Group's Common Object Request Broker Architecture could be integrated in a similar fashion as Python interfaces to such systems become available**.**

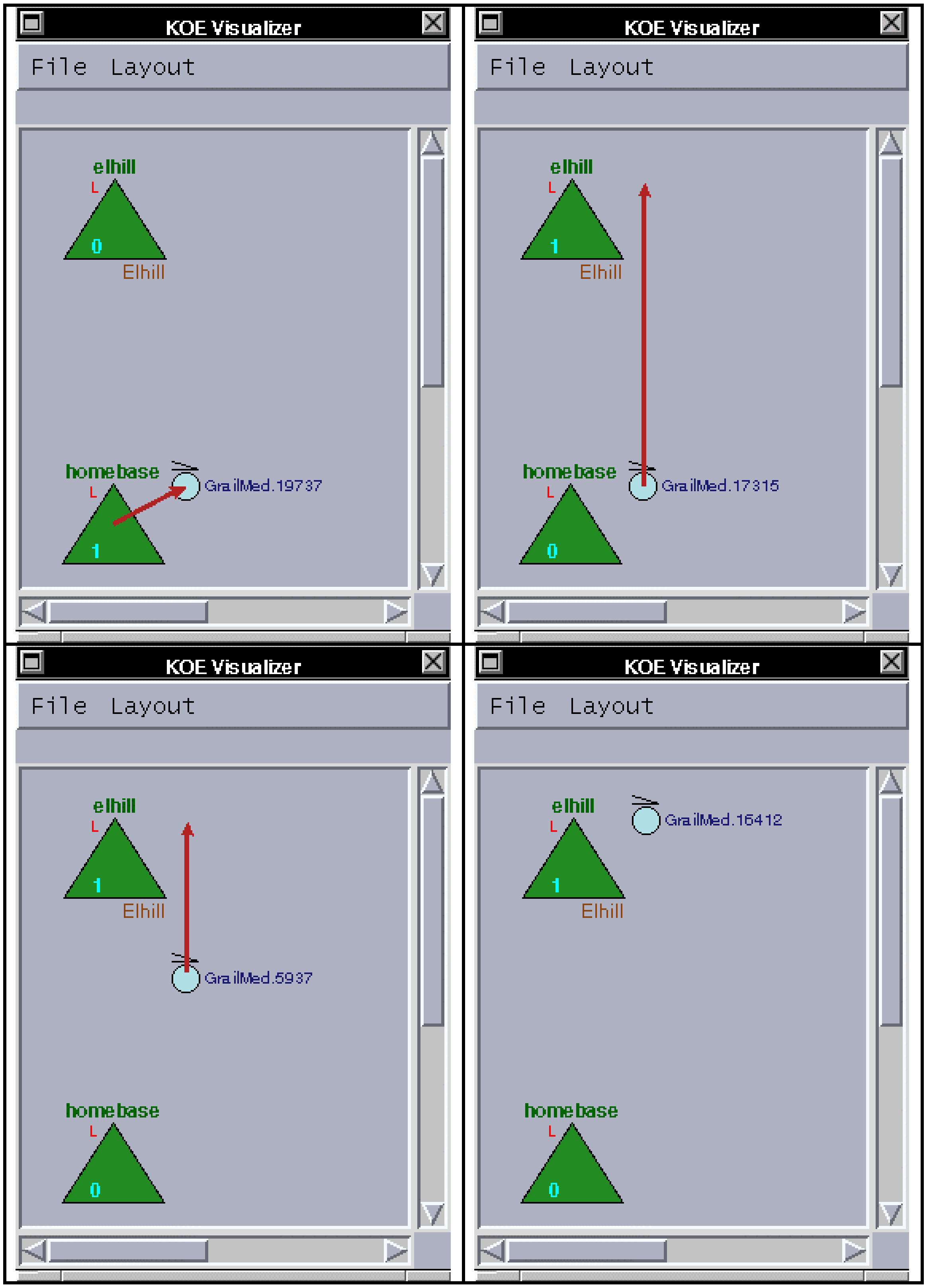

KOE Visualizer
File Layout
elhill
L
0
Elhill
homebase
L
1
GrailMed.19737
KOE Visualizer
File Layout
elhill
L
1
Elhill
homebase
L
0
GrailMed.17315
KOE Visualizer
File Layout
elhill
L
1
Elhill
GrailMed.5937
homebase
L
0
KOE Visualizer
File Layout
elhill
L
1
Elhill
GrailMed.16412
homebase
L
0

Figure 4. Tracking a Knowbot Program's Progress with a Visualizer

The use of a substantial distributed object system within Grail applets required additional support for determining when the resources used by the applet are no longer needed. Requiring support for object release by the underlying GUI package, Tk in the case of Grail, has not proved effective and is not necessarily sufficient without imposing unnecessary requirements on the applet developer. Such reliance is also prone to bugs in the Tk implementation.

To support a general mechanism for resource release by applets without binding the design or implementation to the particular distributed object system used with Grail, a new discipline was introduced for the applet object itself. See Appendix B for details.

Conclusions and Future Work

Our work this year indicates that the rate.d algorithm for distributing load works, does not deadlock, has no race conditions and effectively responds to all service requests. Experiments under simulated load, and with a simulated service shows that the algorithm distributes load across machines with dissimilar capabilities, and that services can be vended on the basis of rate.d allocation. During the coming year CNRI will demonstrate rate.d vending real services under real loading conditions. Future work will evaluate algorithms and approaches necessary to probe various system metrics (CPU usage, I/O bounds, etc.) and to translate these metrics into the necessary rate.d load unit values.

The work performed in integrating the MEDLINE data into the Handle System and repository has shown that there is little perceived difference, from the user's point of view, whether MEDLINE records are retrieved from the Handle system, or from a repository. In terms of performance and user interface, both mechanisms work equally well. Since the Handle System was not designed to store large records (such as MEDLINE ones), future work should concentrate on storing MEDLINE records in one or more repositories. Unique identifiers, based on the Elhill unique identifiers, are stored in the Handle System along with location information about the set of repositories with the full MEDLINE records.

In addition, this work has shown several advantages to migrating the MEDLINE record data to a combined handle/repository system. Primarily, this approach reduces the overall resource strain on the Elhill systems since the amount of work actually performed on Elhill is reduced to the searching side of the problem. Clients need only connect to Elhill for the amount of time it takes to perform a search, and retrieve a list of hit results. Since these results need only include the unique identifier field (although the subject field can optionally be retrieved for improved user interface), the amount of resource utilization on Elhill is reduced, allowing Elhill to service a greater number of requests. This improvement in throughput need not come at the expense of control over the data by NLM, since NLM can, of course, maintain both the repositories and Local Handle Systems.

In addition, this approach has been shown to improve the user interface, since users now have a choice of which results to view. Instead of having to download the entire hit results, 10 records at a time, the user quickly sees a summary of all hit results, with individual details separately retrievable via the repository. This improves the user's perceived control over the system and also aids in total system throughput.

One implementation option is to use multiple repositories at NLM using a farm of inexpensive workstations. Another alternative is to deploy repositories at selected sites outside of NLM. Yet a third alternative is to distribute the query engine to sites outside NLM. The specific choice will depend on what arrangement NLM is willing to enter into. The technology will support any of these options.

During the next period final design and implementation of the AGS Management System and package structure will be completed. The Management System will include support for a security system designed to work with the AGS configured as an Internet firewall. Further live testing of the rate.d protocol with various services will also be completed.

Appendix A. The ILU Interface Specification Language

The interface specification used to support communication between the monitor process and the applets is shown in Figure 5. Though the monitor supports two distinct types of exported information used by the different applets, the basic pattern of operation is the same for both types. The monitor publishs a single object of the type `StatusServer`. Clients of the monitor are required to locate binding information for this interface via an appropriate name service. To receive information about rate.d operation, the client creates a callback object of type `LimitsClient` or `MetricsClient` and registers via the appropriate method on the monitor's `StatusServer` object. Registration returns an ID with which the client can unregister at a later time, or raises an exception. The exception could be used to enforce a simple security policy based on the identity of the client's

```
(**
 *  Definition of interface by which a client / applet can request
 *  notification of changes in AGS status as they occur.
 *
 *  $Revision: 1.12 $
 *  $Source: /projects/cvsroot/nlmdaxs/docs/html/yearly96.html,v $
 *)
INTERFACE StatusAPI;

TYPE String = ilu.CString;
TYPE RegistrationID = String;

(**
 *  If the client requests information about a host the StatusServer
 *  doesn't know about, this exception is raised.
 *)
EXCEPTION UnknownHostError : String;

TYPE MetricItem = RECORD
        (**
         *  Each system metric is forwarded to the client as an object of
         *  this type.  Sequences of several of these pertaining to a
         *  single host are forwarded at one time.
         *)
          smi : INTEGER,
          smd : String
        END;

TYPE MetricSeq = SHORT SEQUENCE OF MetricItem;

TYPE StatusServer = OBJECT
        DOCUMENTATION
          "Server interface which allows registration of various callback
           object with the StatusPublisher which actually implements this
           stuff."

        METHODS
          RequestLimits(callback: LimitsClient) : RegistrationID
            "Allows a client to register a callback object which receives
             notifications of changes in the administrative limitations
             imposed on rate.d hosts.",
           .
           .
           .
```

Figure 5 - Interface definition for monitor communications

location or the identity of the user running the client. The monitor then uses the callback objects registered by clients to pass information as it changes. When the client no longer needs information being provided through the callback interface, it may use the ID returned from the registration to inform the monitor to remove it from the set of callback objects.

The ILU ISL was also used to specify the plugin interface for Elhill. Each object defined in the ISL (bottom 30% of figure 6) contains methods which take specifically typed arguments and return specifically typed values. The first 25% of the ISL file is mainly comprised of exception definitions (many omitted in the figure). These are the various error conditions that can occur while querying the Elhill interface. Following exceptions are the definitions of some specific record types returned by Elhill: `ShortRec` and `LongRec`. Finally, towards the bottom of the file, we see the MEDLINE object definition. It defines a number of methods. These are: `connect`, `close`, `search`, `fetchRecord`, and `fetchUITI`.

```
(* Preliminary interface specification for an Elhill communications KOS extension
 *
 * Recent changes:
 * $Version$
 * $Source: /projects/cvsroot/nlmdaxs/docs/html/yearly96.html,v $
 *)
INTERFACE ElhillController;

EXCEPTION BadService : String
        "Invalid service specified";

EXCEPTION BadDatabase : String
        "Invalid database specified";
        .
        .
        .
TYPE LOGIN = SHORT SEQUENCE OF String;
TYPE LOGINS = SHORT SEQUENCE OF LOGIN;
TYPE SearchTerm = RECORD
        negated : BOOLEAN,
        term : String,
        field : String
END;
TYPE ANDExpression = SEQUENCE OF SearchTerm;
TYPE ORExpression = SEQUENCE OF ANDExpression;
TYPE SearchResult = RECORD
        Exceptional : INTEGER,
        NumFound : INTEGER
END;

TYPE LongRec = RECORD
        TI : String,
        AU : String,
        AD : String,
      .
      .
      .
        Exceptional : INTEGER
END;

TYPE ShortRec = RECORD
        TI : String,
        UI : String,
        Exceptional : INTEGER
END;

TYPE MEDLINE = OBJECT

        METHODS

        connect(Logins : LOGINS) : BOOLEAN
          RAISES LoginFailed, BadService, BadDatabase END
          "Connect to the database",
     .
     .
     .
```

Figure 6 - Interface Definition file for the Elhill plugin

Appendix B. The GrailMED Applet

The GrailMED applet presents a series of Entry fields: Title, Author, Subject, ... that obtain search criteria from the user. The interface allows for terms to be considered individually or together. If a user typed "*Lindberg D*" in the Author Field and "*computer*" in the subject field the terms would be grouped and the Elhill search would be formulated in such a way so that only records that meet both criteria are chosen. With GrailMED, the search is always done on Elhill against the MEDLINE data set. Some experimenting has been done to extend this architecture to other database services offered by Elhill.

Once the search terms are entered in the appropriate fields, the GrailMED user clicks the Search button. This action brings up a new top level window that is designed to obtain a valid MEDLARS username and password from the user. The password entry displays asterisks in place of the actual characters associated with the keys being pressed to assist in keeping the password from being revealed to an onlooker.

Retrieval sources have their own set of controls divided into two major groups: Handle and Elhill. The Elhill method retrieves the first ten records into the applet and the user can scroll through the records. Once the records are pulled over, they are in the applet's memory and no further resolution is needed to view the records. Figure 7 is screen shot of an a GrailMED eperimental window with nine records retrieved from an Elhill search, five of which are shown as displayed on the same page as the applet itself. One need only scroll the display to see the other five. Because the full records are now in the applets memory, further resolution is not needed. The user may page through the records by using the forward and backward arrows.

The Elhill retrieval method also has a cache button that allows retrieved records to be formatted for automatic batch submission to the handle server (with proper authority). Using the caching capability, gradual population of the handle server namespace with MEDLINE records is possible where the most frequently accessed records get uploaded first. Recent versions of the Handle system allow for direct management of handle records, authorities, and other attributes from a program. A next-generation of caching would allow transparent migration of MEDLINE records to handle servers.

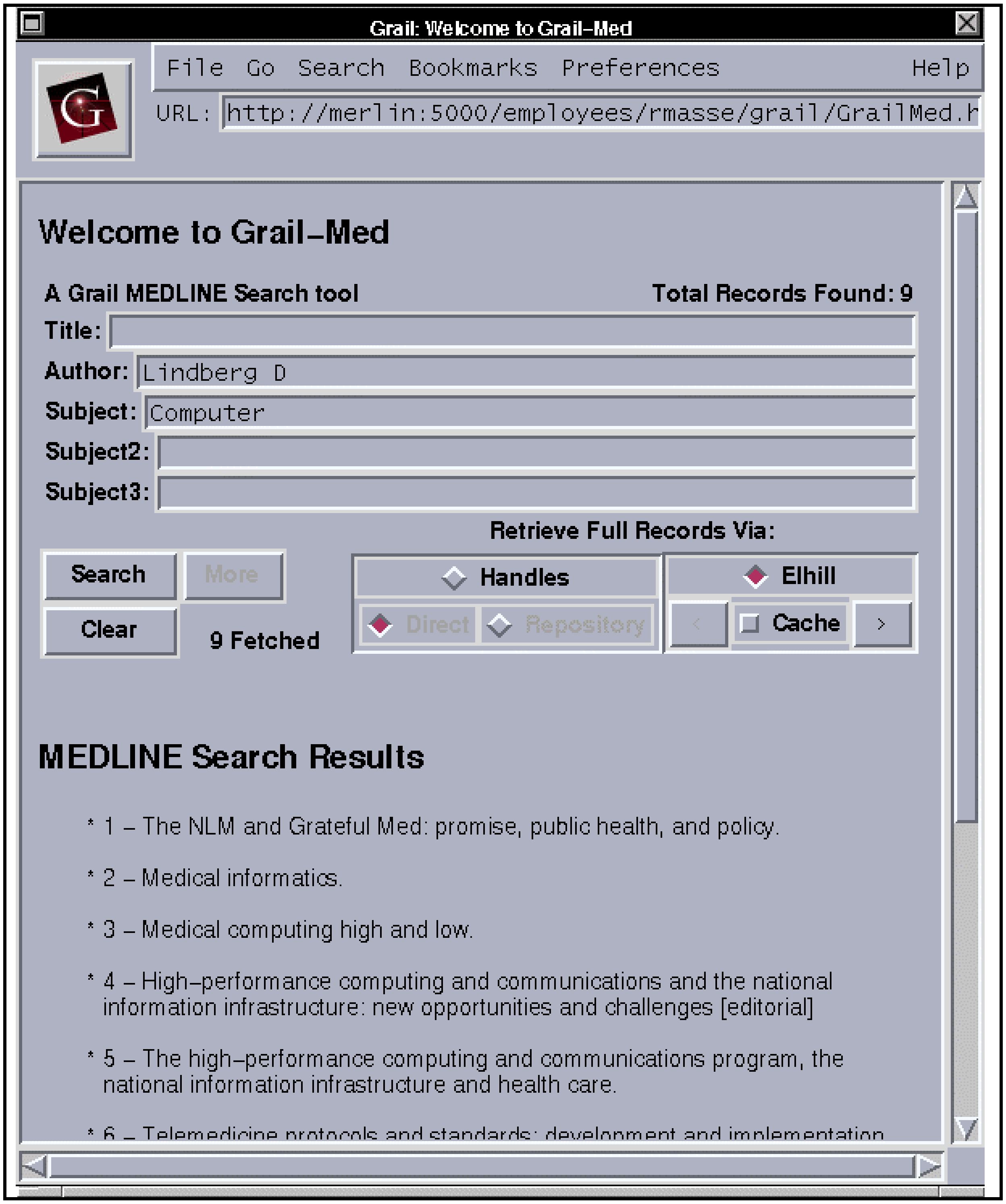
Retrievals using the Handle method are given the further choice of selecting a Repository retrieval or a Direct retrieval. The summary page, in either case, becomes filled with live anchors pointing to handles once the Handle radiobutton is selected. Further, the anchors themselves get dynamically rewritten to specify type=URL for Repository retrievals and type=NLM for direct handle server retrievals. Clearly this many choices presented to a typical user is potentially confusing. Our intent was to demonstrate the flexibility of tools that can separate the searching and retrieval of records thereby allowing storage of records in repositories.

If a search is performed from the start with the Handle radiobutton selected, only the unique identifier and title fields are retrieved from Elhill. These are the only pieces of information necessary to construct the handle and present the user with some meaningful summary results. Figure 8 shows a MEDLINE record that was retrieved from the handle system.

To support a general mechanism for resource release by applets without binding the design or implementation to the particular distributed object system used with Grail, a new discipline was introduced for the applet object itself.

In Python, a discipline is a special protocol used by the runtime support to provide flexible implementations of runtime-controlled events. Disciplines are implemented by object methods with two leading and two trailing underscores.

The __cleanup__() discipline is triggered when the viewer containing the applet is reset in preparation for loading a new page. The method implementing this discipline on the applet requires no parameters other than self. When the applet is loaded, a reference to the discipline method is kept if it is provided by the applet, and called when the viewer is reset. No special action is taken if the discipline is not supported by the applet. Figure 9 shows an example implementation of the applet cleanup discipline.


Grail: Welcome to Grail–Med
File Go Search Bookmarks Preferences Help

URL: http://merlin:5000/employees/rmasse/grail/GrailMed.h

Welcome to Grail–Med

A Grail MEDLINE Search tool

Total Records Found: 9

Title:

Author: Lindberg D

Subject: Computer

Subject2:

Subject3:

Retrieve Full Records Via:

9 Fetched

Handles [✓ Elhill]

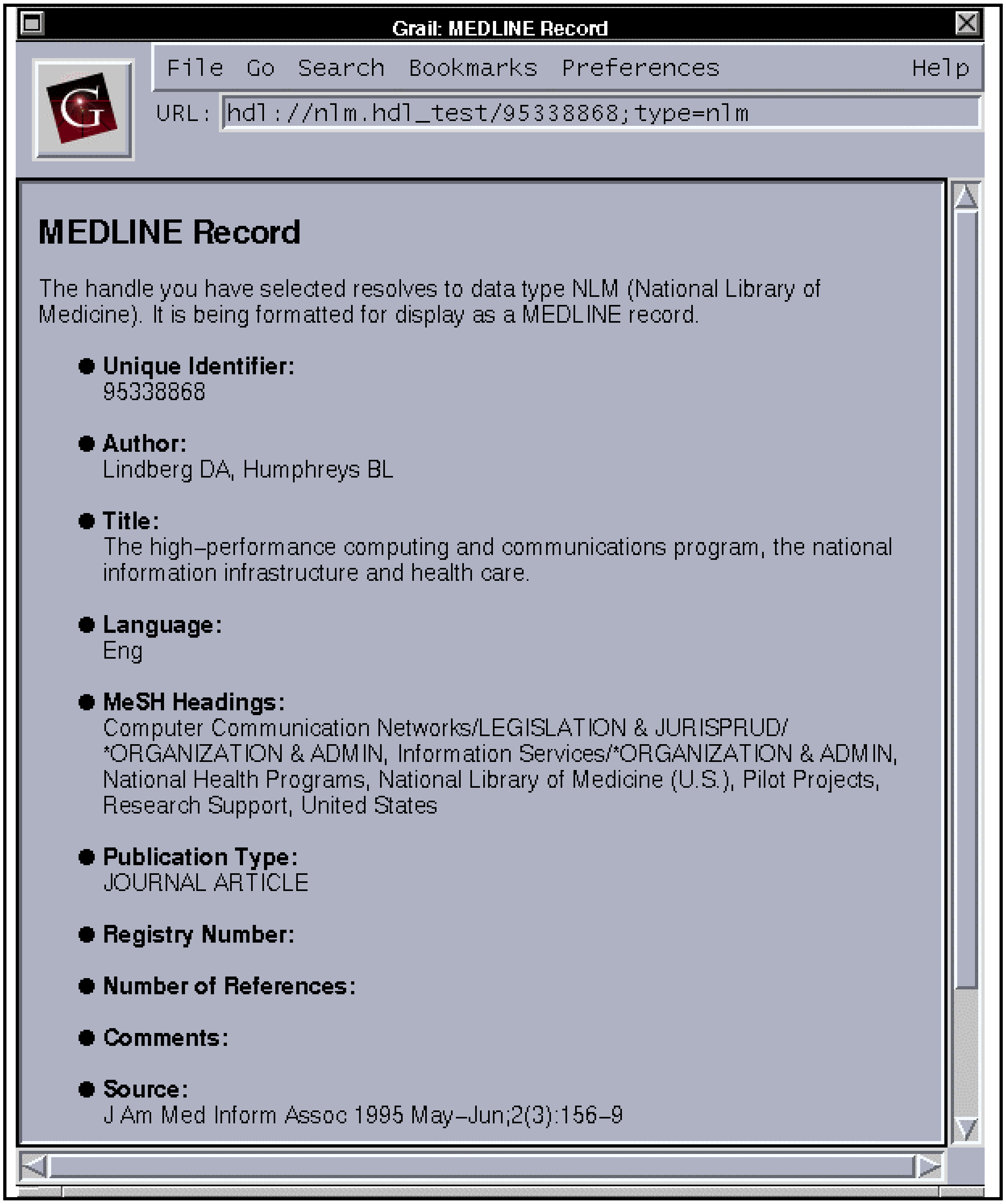
[✓ Direct] Repository

☐ Cache

MEDLINE Search Results

* 1 – The NLM and Grateful Med: promise, public health, and policy.
* 2 – Medical informatics.
* 3 – Medical computing high and low.
* 4 – High–performance computing and communications and the national information infrastructure: new opportunities and challenges [editorial]
* 5 – The high–performance computing and communications program, the national information infrastructure and health care.
* 6 – Telemedicine protocols and standards: development and implementation

Figure 7 - GrailMED Search and Retrieval both via Elhill



hdl://nlm.hdl_test/95338868;type=nlm

MEDLINE Record

The handle you have selected resolves to data type NLM (National Library of Medicine). It is being formatted for display as a MEDLINE record.

- **Unique Identifier:**
  95338868
- **Author:**
  Lindberg DA, Humphreys BL
- **Title:**
  The high-performance computing and communications program, the national information infrastructure and health care.
- **Language:**
  Eng
- **MeSH Headings:**
  Computer Communication Networks/LEGISLATION & JURISPRUD/ *ORGANIZATION & ADMIN, Information Services/*ORGANIZATION & ADMIN, National Health Programs, National Library of Medicine (U.S.), Pilot Projects, Research Support, United States
- **Publication Type:**
  JOURNAL ARTICLE
- **Registry Number:**
- **Number of References:**
- **Comments:**
- **Source:**
  J Am Med Inform Assoc 1995 May-Jun;2(3):156-9

Figure 8 - Retrieving a MEDLINE record from the Handle System

```
# ILU stubs:
import CallbackObject
import CallbackObject__skel

# This class implements an object that another process can call.
Class Applet(CallbackObject__skel.CallbackObject):
    def __init__(self, master, address=None):
        self.connection = None
        self.id = None
        if address:
            self.connection = self.connect(address)
        if self.connection:
            self.id = self.connection.Register(self)

    def __cleanup__(self):
        if self.id:
            self.connection.Unregister( self.id)
            self.connection = None
            self.id = None

    def Event(self, data):
        # callback method
        print "Received", data
```

Figure 9 - Example Cleanup Discipline in an Applet Class

Appendix C. GrailMED Interface to support Knowbot Programs

The GrailMED applet was modified to support an interface for the purpose of constructing, editing, maintaining, launching and receiving information about searching and retrieval via Knowbot Programs. The now familiar top level GrailMED applet window gets two new radiobuttons immediately below the search term entry fields. These are *Direct* and *Knowbot*. Direct corresponds to the default search mechanism outlined previously. Selecting the Knowbot button alters the Search button to read Launcher and causes GrailMED to use the Knowbot Program based mechanisms described below.

Pressing the Launcher button brings up a new top level window that will temporarily be on top of the Grail application.

The launcher window is roughly divided into three sections. At the top is a text field labeled Status Buffer. This is the area where all of the Knowbot Program output is displayed, highlighted in blue. Highlighted in red will be all error conditions including tracebacks of error conditions that caused the Knowbot program to fail. Finally, all normal output written to the Knowbot Programs standard output will be displayed in black.

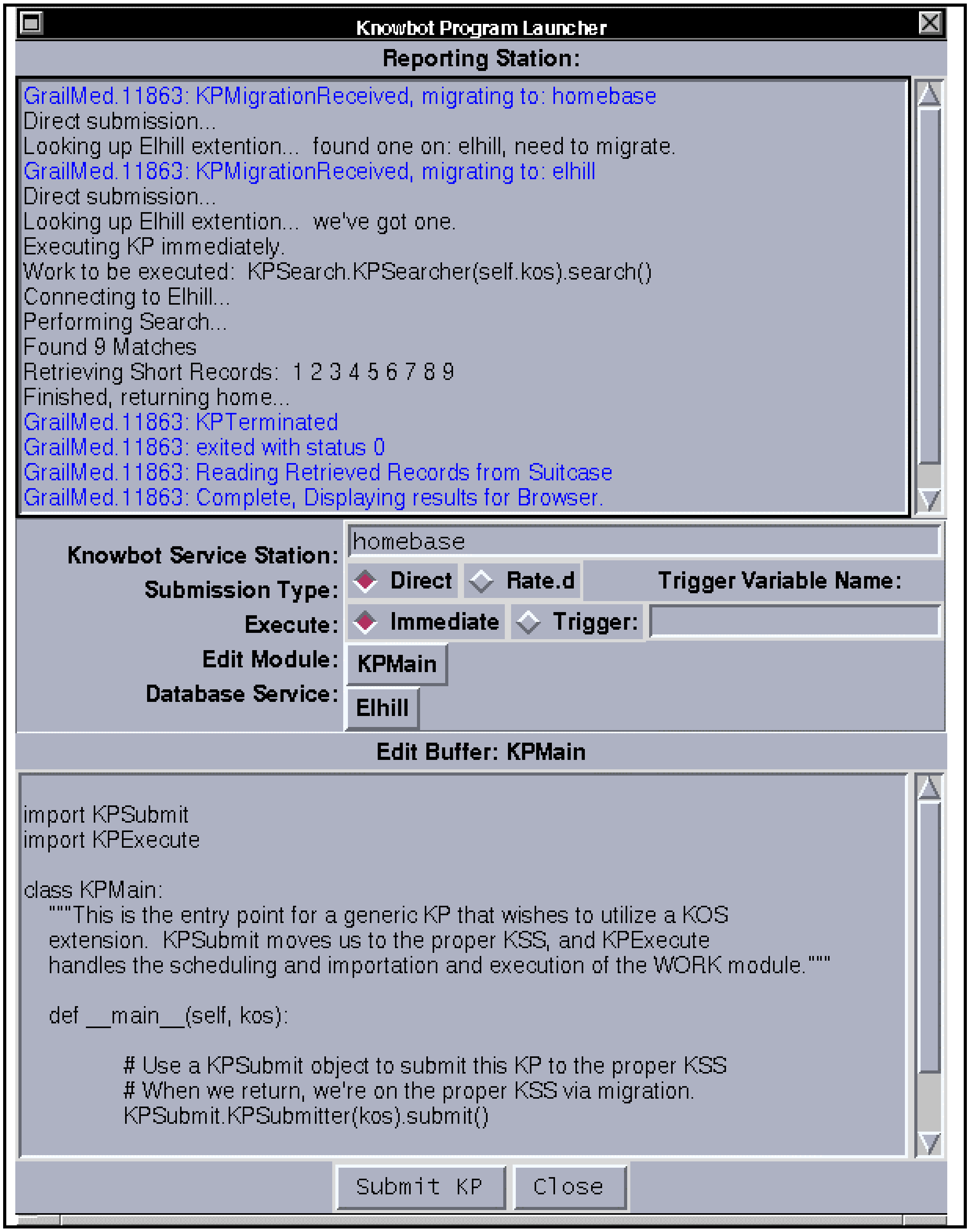
The bottom section of the launch panel is the edit buffer. This is the area which displays and maintains the state of the various modules comprising the Knowbot Program. In the case of this applet the Knowbot Program is comprised of five separate modules.

- KPMain - Provides the entry point for a generic Knowbot Program. It calls into the KPSubmit and KPExecute modules to initiate the appropriate type of submission and execution.
- KPSearch - Provides the functions which will conduct the search and store the results in the Knowbot Programs Suitcase for return. The suitcase is one way for Knowbot Programs to carry around information they wish to retain.
- KPExecute - Handle the scheduling of when the Knowbot Program should run, immediately or based on a trigger.
- KPSubmit - Handle the method of submission (currently only 'direct' is implemented). In addition, locate the Elhill plugin and migrate there if necessary.
- KPVars - The module where all of the non-boilerplate state is stored. Examples are: login information, search terms, execution type, submission type, and Elhill interface information.

The center portion of the launcher panel contains the configuration controls. Manipulation of these controls rewrites part of the information stored in the KPVars module. A summary of the controls follows:

- Knowbot Service Station: The KSS that will serve as the first stopping point for the Knowbot Program.
- Submission Type: Either direct or rate.d are possible here. Only direct is currently implemented.
- Execute: Either Immediate or Trigger are possible here. If trigger is selected Trigger Variable Name: will indicated the trigger variable this Knowbot Program will wait for to fire before conducting the search.
- Edit Module: This is a menu button that when pressed reveals a menu of Modules that are the parts of the Knowbot Program. Selecting any of these modules brings that module into the Edit Buffer.
- Database Service: This is a menu button that when pressed will reveal the various database services the Knowbot Program can connect to. Currently there is only one selection, Elhill.

Pressing the Submit button on the very bottom of the launcher panel will conduct the same search we demonstrated in a previous section except that the details of the search properties and the results of the search are transported and represented in a Knowbot Program. Figure 10 shows the Launcher window and some typical reporting station output that might be printed during a search.



```
GrailMed.11863: KPMigrationReceived, migrating to: homebase
Direct submission...
Looking up Elhill extention...  found one on: elhill, need to migrate.
GrailMed.11863: KPMigrationReceived, migrating to: elhill
Direct submission...
Looking up Elhill extention...  we've got one.
Executing KP immediately.
Work to be executed:  KPSearch.KPSearcher(self.kos).search()
Connecting to Elhill...
Performing Search...
Found 9 Matches
Retrieving Short Records:  1 2 3 4 5 6 7 8 9
Finished, returning home...
GrailMed.11863: KPTerminated
GrailMed.11863: exited with status 0
GrailMed.11863: Reading Retrieved Records from Suitcase
GrailMed.11863: Complete, Displaying results for Browser.
```

Knowbot Service Station: homebase
Submission Type: [✓ Direct] Rate.d
Trigger Variable Name:
Execute: [✓ Immediate] Trigger:
Edit Module: KPMain
Database Service: Elhill

```
import KPSubmit
import KPExecute

class KPMain:
    """This is the entry point for a generic KP that wishes to utilize a KOS
    extension.  KPSubmit moves us to the proper KSS, and KPExecute
    handles the scheduling and importation and execution of the WORK module."""

    def __main__(self, kos):

            # Use a KPSubmit object to submit this KP to the proper KSS
            # When we return, we're on the proper KSS via migration.
            KPSubmit.KPSubmitter(kos).submit()
```

Figure 10- The Knowbot Program Launcher Panel during a Search