Annual Summary
Application Gateway System

NSF Grant Number: NCR-9302522

January 1,1995 - December 31, 1995

Corporation for National Research Initiatives

1895 Preston White Drive, Suite 100

Reston, VA 20191

Summary

This report discusses the progress made on NSF Grant NCR-9302522 during calendar year 1995. It provides an updated description of a scaleable high performance application gateway system that will support reliable access for a large number of network requests to existing and future database repositories. Access to disparate database and other services are consolidated in a uniform access mechanism provided by the gateway system. These gateway platforms act as a security minded intermediary between the Internet and a broad range of back-end repositories present on many of today's private internal networks. In 1995, we completed an initial implementation of the LAN-based load balancing algorithm known as rate.d. future work involves fine tuning of the algorithm and extending it to work on WAN based AGS systems.

The Application Gateway System (AGS) provides a service called the Distributed Application eXtension Service (DAXS). The AGS consists of a distributed system of replicated servers (of the same or differing performance) that collectively provide highly available access to database resources. Users request services from the AGS without knowledge of which processor is going to service their request. The AGS dynamically selects the best processor at the time, submits the users request to that processor, executes the job and returns the result(s). These servers accomplish this distribution of processing by running a copy of a load balancing server daemon known as rate.d which is used to apportion the load among the processors.

The AGS provides a highly reliable service because of the replicated processors none of which individually are required to remain in operation, provide for continuous operation as long as at least one is operating. Further reliability can be achieved by configuring the AGS with dual routers that are configured to serve as backup for each other in case of failure. Additional AGS providing the same service can be geographically dispersed and operated simultaneously to provide greater reliability.

The need for a new load balancing algorithm was realized very early in this project. Existing Internet mechanisms to distribute load, (e.g. round robin DNS), fail because of the use of client side caching to speed the delivery of the information about which address to use. Even if the caching problem did not exist, round robin DNS simply sequences through a list of servers without any regard for the capabilities or load of each server. The need was recognized to implement a load balancing algorithm which would consider server capabilities and load while avoiding response storms from large collections of servers. The initial effort on this project therefore concentrated on specifying and implementing this new algorithm.

Substantial progress was made toward the overall system design and implementation of the AGS and a demonstration of several AGS components was carried out. Progress was made in the areas of language selection, database interface, security framework, control interface and the distribution of processing.

Application Gateway System

Overview

A configuration of an AGS with interfaces to databases and repositories on the back-end is shown in figure 1. The AGS is comprised of one or more loosely coupled processors connected via a Local Area Network technology, such as Ethernet. Each machine that is a member of an AGS is termed an Application Gateway (AG) and participates in the AGS by virtue of three important facts: 1) Each AG runs application services to meet specific user needs. 2) Each AG mirrors the capabilities of its sister machines (i.e. except for hardware and OS specifics, each AG is a clone of the others). 3) Each AG participates in a load balancing service known as rate.d. Rate.d is discussed in more detail in subsequent sections.

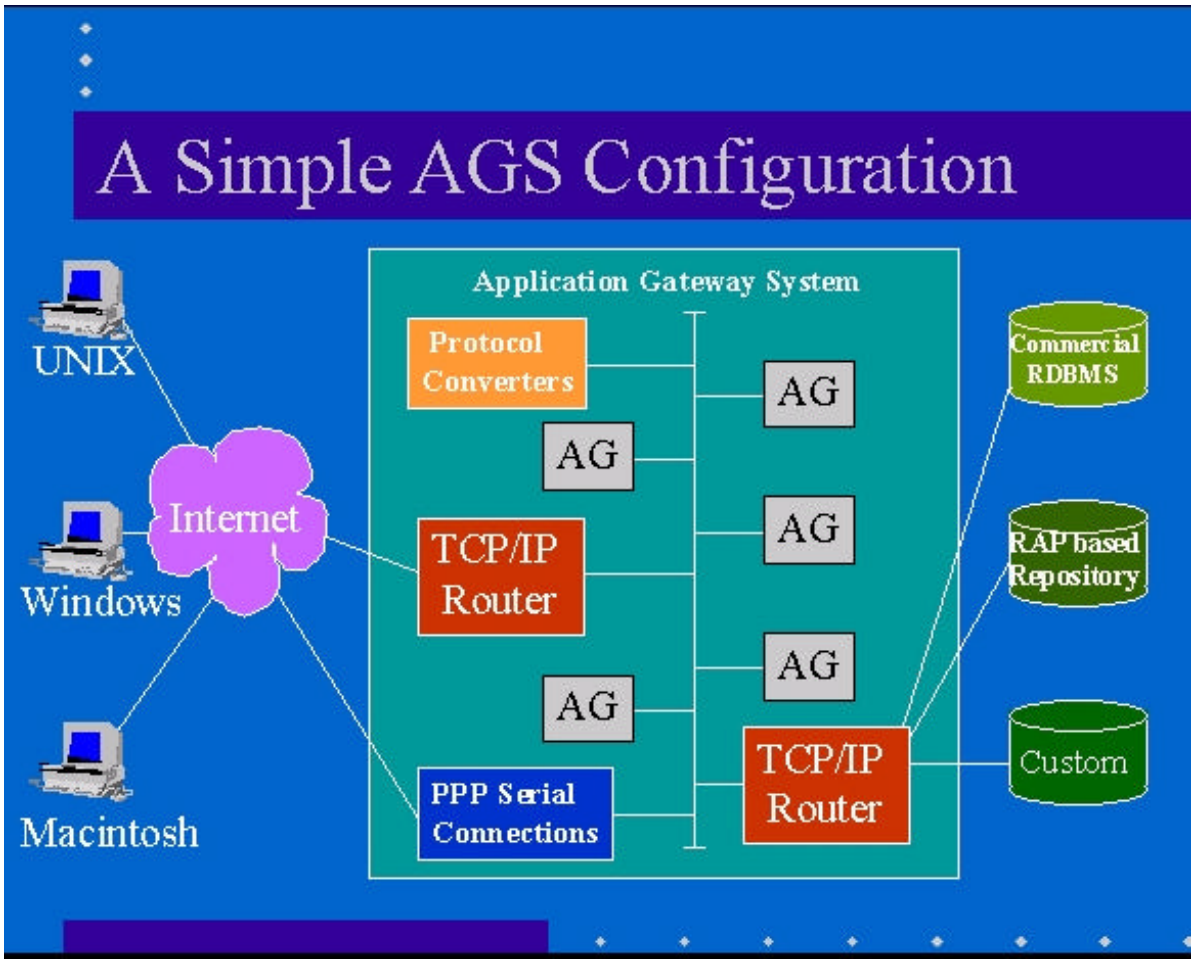


Figure 1 - Configuration of the Applications Gateway System

AGS Design & Implementation

The design philosophy for the AGS is based on a system that is highly reliable, provides security via firewall technology, and can also scale according to demand for the information. The reliability and scalability is accomplished by providing a number of mirrored servers called Application Gateways that share demand for their resources. By having more than one AG available, the system provides redundancy and permits multiple, simultaneous requests for service. For example, a large farm of FTP servers would clearly benefit from the capabilities of an AGS by allowing only the least utilized server to respond to a request for service. By contrast, in other implementations, the *user* must choose from a list of FTP servers with unknown loads. The AGS design addresses the balancing of load between servers. Firewall technology consisting of front and back end filtering routers which force access through a set of trusted application gateways, provide the bases of security which is described below.

The AGS recognizes requests for service as *jobs*. The AGS does not currently attempt to share the execution of a single job between multiple AGs although this feature could easily be added on a static or dynamic basis. Rather, the AGS delegates an entire job to a single AG.

Much of an AGs participation in the AGS is automatic. Specifically, when an AG comes on-line, the other AGs are made aware of this and begin sharing jobs with it. Similarly, if an AG becomes unavailable, for some reason, the others become aware of this AGs absence by its lack of participation. When this happens, the other AGs take up the entire load in its absence.

An AGS can be configured as an Internet Firewall. To facilitate this, both routers in figure 1 are capable of filtering IP packets based on: source address, destination address, IP service and the direction from which the connection is originated. The router between the AGS and the Internet simply filters out all unwanted services - for example, NFS, X, TFTP, BOOTP, and other services which could compromise system integrity. The router between the AGS and the protected network only allows connections to and from the AGs. As a result, direct connectivity from the Internet to the private network is not allowed. All connections must be *relayed* through an AG. To provide security on each AG, only the needed services are enabled. Known *dangerous* services are explicitly prevented from running. In addition, all connections are logged and these log entries are forwarded, as they are entered, to a *safe* host within the protected network.

CNRI was originally planning on implementing the majority of the AGS software in Objective-C and using commercial database tools as the platform for the database interface. However, CNRI settled on using the Python as the implementation language. Python is a very-high-level object-oriented scripting language that is extensible. Python is "modules" based and it "imports" only the code that is necessary to run a particular program, which keeps the interpreter environment clean and uncluttered. For additional speed and efficiency, where specifically needed, Python can be extended in C or C++ quite easily by using a thin layer of Python "glue" over the C or C++ to make the Python interpreter aware of the new extension module. Python can also be embedded in C or C++ and used to parse configuration information for a larger software system. Python also runs on virtually any platform available today.

Python was made available to NLM, and members of their staff were trained on the use of Python. A seminar series consisting of six seminars on the Python Programming Language was given at CNRI. This series was videotaped, and copies of these tapes are available.

Adding New Services to the AGS

Adding a new service to the AGS requires it to be closely integrated with the load balancing implementation called rate.d discussed later in this report. Currently, each type of service which has it's own load characteristics should be managed separately; Each separately managed service requires it's own set of rate.d server daemons running on a port specific to that service. This restriction should be lifted in the future.

Assuming a service is enabled to the point where it could normally operate in a single system environment, the service needs to do three additional things in order to participate in AGS load balancing.

- The service needs to have the Job ID (JID) which it gets from the client (The client gets JID from rate.d during the service request phase, see Section on Distribution of Processing)
- The service needs to send a Job Execute Start (JXS) message to the rate.d process managing this server to inform rate.d that job processing has begun.
- The service needs to send a Job Execute End (JXE) message to the rate.d process managing this server, once the job has completed.

Database Interface

Database interfaces are a key part of the AGS. The AGS provides a uniform database interface to typical commercial packages such as Oracle, Sybase or Informix and also for non-relational or non-tabular databases as well. The Elhill interface to Medlars and the repository interface developed at CNRI are two examples of interfaces that the system supports. Other important database interfaces would include NLM databases on AIDS, Cancer, Dentistry, Genetics, Population Studies, Space Medicine, Toxicology, Environmental Health and Bioethics.

Generalized Tabular Database Interface

A generalized form of database interface standardization in Python is provided by the system. To facilitate this, a database Special Interest Group was formed among the Python users for the purposes of developing a relational/tabular database interface for Python . The preliminary interface specification was developed in late 1995 and is described in the next section. The NLM Elhill and CNRI repository interfaces are being modeled within this framework, to the extent that is reasonable for these non-tabular databases.

This API has been defined to encourage similarity between the Python modules that are used to access databases. By doing this, we hope to achieve a consistency leading to more easily understood modules, code that is generally more portable across databases, and a broader reach of database connectivity from Python.

This interface specification consists of several items:

- Module Interface
- Connection Objects
- Cursor Objects
- DBI Helper Objects

Further information is available via: *http://www.python.org*

Module Interface

The database interface modules should typically be named with something terminated by db. Existing examples are: oracledb, informixdb, and pg95db. These modules should export several names:

modulename(connection_string)
Constructor for creating a connection to the database. Returns a Connection Object.

error
Exception raise for errors from the database module.

Connection Objects

Connections Objects should respond to the following methods:

close()
Close the connection now (rather than whenever __del__ is called). The connection will be unusable from this point forward; an exception will be raised if any operation is attempted with the connection.

commit()
Commit any pending transaction to the database.

rollback()
Roll the database back to the start of any pending transaction.

cursor()
Return a new Cursor Object. An exception may be thrown if the database does not support a cursor concept.

callproc([params])
Note: this method is not well-defined yet.
Call a stored database procedure with the given (optional) parameters. Returns the result of the stored procedure.

Cursor Objects

For databases that do not have cursors and for simple applications that do not require the complexity of a cursor, a Connection Object should respond to each of the attributes and methods of the Cursor Object. Databases that have cursor can implement this by using an implicit, internal cursor.

These objects represent a database cursor, which is used to manage the context of a fetch operation.

Cursor Objects should respond to the following methods and attributes:

arraysize
This read/write attribute specifies the number of rows to fetch at a time with fetchmany(). This value is also used when inserting multiple rows at a time (passing a tuple/list of tuples/lists as the params value to execute()). This attribute will default to a single row.

Note that the arraysize is optional and is merely provided for higher performance database interactions. Implementations should observe it with respect to the fetchmany() method, but are free to interact with the database a single row at a time.

description
This read-only attribute is a tuple of 7-tuples. Each 7-tuple contains information describing each result column: (name, type_code, display_size, internal_size, precision, scale, null_ok). This attribute will be None for operations that do not return rows or if the cursor has not had an operation invoked via the execute() method yet.

The type_code is one of the dbi values specified in the section below.

Note: this is a bit in flux. Generally, the first two items of the 7-tuple will always be present; the others may be database specific.

close()
Close the cursor now (rather than whenever __del__ is called). The cursor will be unusable from this point forward; an exception will be raised if any operation is attempted with the cursor.

execute(operation [,params])
Execute (prepare) a database operation (query or command). Parameters may be provided (as a sequence (e.g. tuple/list)) and will be bound to variables in the operation. Variables are specified in a database-specific notation that is based on the index in the parameter tuple (position-based rather than name-based).

The parameters may also be specified as a sequence of sequences (e.g. a list of tuples) to insert multiple rows in a single operation.

A reference to the operation will be retained by the cursor. If the same operation object is passed in again, then the cursor can optimize its behavior. This is most effective for algorithms where the same operation is used, but different parameters are bound to it (many times).

For maximum efficiency when reusing an operation, it is best to use the setinputsizes() method to specify the parameter types and sizes ahead of time. It is legal for a parameter to not match the predefined information; the implementation should compensate, possibly with a loss of efficiency.

Using SQL terminology, these are the possible result values from the execute() method:

If the statement is DDL (e.g. CREATE TABLE), then 1 is returned.
If the statement is DML (e.g. UPDATE or INSERT), then the number of rows affected is returned (0 or a positive integer).
If the statement is DQL (e.g. SELECT), None is returned, indicating that the statement is not really complete until you use one of the fetch methods.

fetchone()

Fetch the next row of a query result, returning a single tuple.

fetchmany([size])

Fetch the next set of rows of a query result, returning as a list of tuples. An empty list is returned when no more rows are available. The number of rows to fetch is specified by the parameter. If it is None, then the cursor's arraysize determines the number of rows to be fetched.

Note there are performance considerations involved with the size parameter. For optimal performance, it is usually best to use the arraysize attribute. If the size parameter is used, then it is best for it to retain the same value from one fetchmany() call to the next.

fetchall()

Fetch all rows of a query result, returning as a list of tuples. Note that the cursor's arraysize attribute can affect the performance of this operation.

setinputsizes(sizes)

Note: this method is not well-defined yet.
This can be used before a call to execute() to predefine memory areas for the operation's parameters. Sizes is specified as a tuple -- one item for each input parameter. The item should be a Type object that corresponds to the input that will be used, or it should be an integer specifying the maximum length of a string parameter. If the item is None, then no predefined memory area will be reserved for that column (this is useful to avoid predefined areas for large inputs).

This method would be used before the execute() method is invoked.

setoutputsize(size [,col])

Note: this method is not well-defined yet.
Set a column buffer size for fetches of large columns (e.g. LONG). The column is specified as an index into the result tuple. Using a column of None will set the default size for all large columns in the cursor.

This method would be used before the execute() method is invoked.

DBI Helper Objects

Many databases need to have the input in a particular format for binding to an operation's input parameters. For example, if an input is destined for a DATE column, then it must be bound to the database in a particular string format. Similar problems exist for "Row ID" columns or large binary items (e.g. blobs or RAW columns). This presents problems for Python since the parameters to the execute() method are untyped. When the database module sees a Python string object, it doesn't know if it should be bound as a simple CHAR column, as a raw binary item, or as a DATE.

To overcome this problem, the dbi module was created. This module specifies some basic database interface types for working with databases. There are two classes: dbiDate and dbiRaw. These are simple container classes that wrap up a value. When passed to the database modules, the module can then detect that the input parameter is intended as a DATE or a RAW. For symmetry, the database modules will return DATE and RAW columns as instances of these classes.

A Cursor Object's description attribute returns information about each of the result columns of a query. The type_code is defined to be one of five types exported by this module: STRING, RAW, NUMBER, DATE, or ROWID.

The module exports the following names:

dbiDate(value)
This function constructs a dbiDate instance that holds a date value. The value should be specified as an integer number of seconds since the "epoch" (e.g. time.time()).

dbiRaw(value)
This function constructs a dbiRaw instance that holds a raw (binary) value. The value should be specified as a Python string.

STRING
This object is used to describe columns in a database that are string-based (e.g. *CHAR*).

RAW
This object is used to describe (large) binary columns in a database (e.g. LONG RAW, blobs).

NUMBER
This object is used to describe numeric columns in a database.

DATE
This object is used to describe date columns in a database.

ROWID
This object is used to describe the "Row ID" column in a database.

NLM Elhill Database Interface

An extension module has been implemented in Python that allows Python programs to conduct searches against the MEDLINE database. The interface allows the program to conduct a search based on a search expression. The MEDLINE Interface object consists of five methods which act upon it. These are defined in a CORBA-like Interface Definition Language below:

```
TYPE MEDLINE = OBJECT

    METHODS

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

    close()
      "Closes the database connection",

    search(Expression : ORExpression) : SearchResult
      RAISES LoginFailed, BadTerm,
          BadService, BadDatabase END
      "Perform an Elhill search based on SearchExpression",

    fetchRecord(RecNum : INTEGER) : LongRec
      RAISES LoginFailed, UnexpectedLine, NoFields,
          BadService, BadDatabase END
      "Fetch a single record from a search",

    fetchUITI(RecNum : INTEGER) : ShortRec
      RAISES LoginFailed, NoUI, NonNumericUI, NoTitle,
          BadService, BadDatabase END
            "Fetch a Unique ID  and Title  for a single record from a search"

    END;
```

The Search expression

The search expression is defined as a Python list object containing Python lists containing Python dictionaries. The top level tuples are divided on either side of an OR operation, the next level joining lists in AND, Finally the dictionary is a triple containing ('not', 'term', 'field') as keys. Thus, an expression containing:

```
(({'not' : 0, 'term' : 'lung', 'field' : None),
  {'not' : 0, 'term' : 'cancer', 'field' : None),
  {'not' : 1, 'term' : 'for', 'field' : '(LA)')),
 {'not' : 0 , 'term' : 'Lindberg d', 'field' : '(AU)'))
```

would evaluate to the following Elhill search:

```
lung and cancer and not for (la) or Lindberg d; (AU)
```

User Interfaces for the AGS

The AGS provides a framework by which ordinary Internet services may be replicated and distributed. Client access to these services on an AGS requires the clients to request the service in accordance to the Rate.d client application protocol which is described in a later section. A client library has been implemented as a convenience for applications programmers interested in distributing services via an AGS. Web browsers such as Netscape and Grail support the downloading of *applets*. Applets can include the AGS library code such that access to the AGS is handled completely transparently.

For support of existing clients that cannot be changed, the AGS can be configured to have a dedicated *proxy* gateway that is rate.d aware and can *relay* requests to the appropriate AG for service.

Grail is an example AGS client. It is an Internet Browser written in Python and being developed with support from DARPA. Currently, Grail can also be used as a framework for communication over the Internet: it can serve as an HTML 2.0 compliant World Wide Web browser, it can interact with repository based systems and it provides an open extensible architecture for supporting a wide range of standard and experimental protocols. Grail also supports the use of applets, small Python programs downloaded from a remote server, which run within the interface.

CNRI plans to use the applet capabilities of the Grail Browser for the majority of the AGS control interfaces. These interfaces include applets that: control the distribution of processing, add or remove database interfaces and launch Knowbot Programs capable of searching repositories local to the AGS.

Service Environments for the AGS

Possible secure environments are provided by World-wide-web servers, FTP servers, and several layers which might be provide by Z39.50. In general, the AGS is able to house a wide class of such servers all running at the same time. The access to the UMLS Metathesaurus was also demonstrated in this fashion.

The Knowbot Service Station (KSS) is another example AGS service component. A KSS is a combination of hardware, system software, and special software that allow the submission of mobile software agents known as Knowbot Programs. The special software is called the Knowbot Operating System, or KOS. The KOS actually consists of a number of cooperating processes running different components of the KOS. Thus, the system hardware and low-level system software must be of sufficient power to support this KOS architecture, and currently this is supported by use of Unix workstations and the Unix operating system. There is one designated process called the KOS kernel (not just "kernel", to prevent confusion with the Unix kernel). Other processes are assigned to running KPs (one process per KP). The code running inside a KP process consists of the KP user code (the actual Knowbot Program that some user wrote to perform a task) and the KP supervisor code. The KP supervisor is considered part of the KOS (though not of the KOS kernel). In the case of a KP written in Python, the KP user code runs in restricted execution mode, and the KP supervisor code runs in unrestricted mode. For other languages, a similar distinction has to be made. Strictly speaking, the Python interpreter and the C run-time library are also executing in the KP process. However, it is not necessary to think about these as separate entities.

Distribution of Processing via Load Balancing

This section describes a protocol for a distributed load balancing algorithm called rate.d (pronounced ``rate-dee''). The rate.d load distribution algorithm has been specified to include the balancing of unlike tasks such as are possible in a Knowbot Operating Environment. In this period rate.d has gone from a conceptual description to an implementation with multiple copies on a LAN, In addition a monitor process has been added to assist in tracking packet flow and algorithm performance. A more detailed description of rate.d is given below.

In the current implementation, the user cannot change the parameters of the rate.d algorithm. Future implementations could provide hooks to allow users to change those parameters which are appropriate for the user to change.

Currently it is necessary to have a client library to utilize services managed by the rate.d protocol. This could present obstacles to programs which are not rate.d enabled. One could also provide mechanisms for such programs to request service from a AGS without the client library. For example, a gateway host could be an intermediary between legacy software and the AGS.

Rate.d

Rate.d's function is to distribute job allocation among similarly configured servers working in coordination within a single local area network. Several servers will typically be running in a multiprocessor configuration on a local area network, communicating with each other as jobs are requested and executed. Incoming jobs can be distributed evenly among the servers, thus providing a higher level of overall system throughput and reliability.

The rate.d algorithm specifically addresses issues of high availability, robustness in system design, and distributed decision processing, such that there is no single point of failure in the algorithm. Rate.d adapts rapidly to changing environmental factors, such as servers crashing, new servers coming on-line, or changing system loads, and the algorithm provides important metrics that administrators can use to monitor the state of the entire cluster. Rate.d minimizes the network traffic between the cluster and the client so as to avoid response storms which could overload mediating networks.

Rate.d specifically does not provide any mechanisms to support service location or selection. Other mechanisms, such as the proposed Service Location Protocol, should be used to locate providers of required services. Rate.d pertains only to the allocation of service provision with a cluster of servers capable of providing similar services but which require controlled load balancing.

Rate.d System Objects

The rate.d algorithm involves the following top-level conceptual objects:

Application Protocol

> The communications protocol or protocol suite by which the client and Application Protocol Daemon conduct transactions. The Application Protocol may be implemented over any available transport, and is not required to operate on the same transport as the rate.d system used to allocate access to the application.

Application Protocol Daemon

The subsystem within a single server that executes Jobs. Every server has exactly one Application Protocol Daemon.

Clients

The remote system that generates and launches requests for execution after submittal and acceptance of the associated Job. There is a theoretically infinite number of clients which can communicate with any cluster.

Service Requests

Individual requests for service provision.

Jobs

Data pertinent to the provision of service which is supplied by the client directly to the Application Protocol Daemon. Each Job may include data, executable code, or both.

Rate.d

The subsystem of a service which determines whether to accept each incoming Service Request. Every server has exactly one Rate.d.

Rater.d

The subsystem that converts local system information into metrics values used in determining job acceptance. Pronounced rater-dee." The discovery of load averages is highly dependent on the operating system and application. For this reason, an API to rater.d is described, while the determination of metrics is left as an implementation detail. Every server has exactly one Rater.d process.

Rate.d Job States

During the time from the submission of a service request through the completion of a job, each rate.d daemon describes the job as being in one of the four states described below.

Unassigned

An initial state assigned to newly received jobs which have not been analyzed for acceptance.

Pending

The state assigned to jobs which are not immediately accepted for execution. Jobs with state “Pending" will usually be accepted by some other server on the local area network.

CommittedTo

The state assigned to jobs for which resources have been allocated, but not consumed, on the designated server.

Running

The state assigned to jobs which represent service requests running on the designated server, consuming resources.

Rate.d System Metrics

The decision as to which system will accept a job is made using the Compulsory Volunteer Algorithm specified for the application being served. This algorithm is discussed in more detail below.

The system metrics used by the Compulsory Volunteer Algorithm will depend to a large degree on both the application and the hardware and software platform running the server. For this reason, the rate.d daemon uses a separate process to provide system metrics for use by the algorithm. The Rater.d system object is this process, and is required to supply to appropriate metrics periodically via the System Metrics Analysis (SMA) message.

Rate.d Algorithm Walk through

The rate.d algorithm is event driven. The process of job acceptance centers around the determination of bestness according to a metric. When a new job request is received at the AGS: the server on the local area network which is best able to accept the request is selected to execute the service request. This decision making process is fully distributed and is always running. It is invoked each time client a transmits a Request for Service (RFS) message to the system.

Initial Contact

The client makes a request to the single published Internet address which is delivered to all the AGs. The address used should be a broadcast or multicast address so that each rate.d process on the local area network receives the incoming request. When a server determines that it is best able to respond to an RFS message it accepts the job and commits resources for the execution of the job. If it determines that it is not the best, it declines to accept the job knowing that some other server will accept the job. This is called pending the job. A pended job is not discarded: the designated server will place a reference to the pended job on a Jobs Pending List. This is done in case the server determined to be best never actually accepts the requested job (e.g. perhaps because it crashed in the interim). This mechanism provides protection against network failures and protocol unreliability by providing secondary points of acceptance. See below for details.

When a job is accepted, the designated server places the job on its Jobs Committed To List and allocates any resources necessary to execute the service request. Note that the job is not actually executed at this time: the details of the job have not yet been transmitted to the server. In addition, the designated server sends two messages informing the rate.d processes associated with local servers that the job has been accepted:

- a Job eXecution Committed To (JXC) message is sent to the client. This informs the client that a server has allocated resources to its service request. This message contains any information necessary for the client to contact the server (e.g., a handle, IP address and a port, a URL, and any necessary authorization ticket). This message is sent, point-to-point, from the rate.d process to the client.

- a System Metrics Analysis (SMA) message is sent to the local area network. This is used for two reasons. First, it informs all the local servers that a server has accepted the job. Second, it updates metric information at all the local servers. This is the way all the servers within the local area network remain synchronized. This message is sent to the same address used by external clients to contact the server cluster initially, and thus takes advantage of whatever broadcast or multicast mechanisms are being employed to distribute messages within the cluster. Since the current metrics are sent rather than change information, messages missed by other servers do not incur serious synchronicity problems. This allows lightweight and unreliable protocols, such as UDP, to be used within a server cluster.

Execution of Service Requests

When a client receives a JXC message from one of the servers, it sends job information to the server, packaged within a Request for Execution (RFE) message. Assuming that the allocation of resources on the server have not timed out (see Resource Allocation Time-out), the job is forwarded to the Application Protocol Daemon (APD). At the same time, a Job eXecution Started (JXS) message is sent to the rate.d daemon which moves the job from the Jobs Committed To list to the Jobs Running List. This prevents rate.d from timing out on the job after it has begun execution.

Completion of Service Requests

When the job has completed executing, irrespective of exit status, a Job eXecution Ended (JXE) message is sent to the rate.d daemon by the APD.

Resource Allocation Time-out

When an server commits resources toward the execution of a client's service request, this resource allocation does not extend for all time. If the client does not submit the execution request to the selected server within some period of time, the resource allocation can time-out. In this case, the server reserving the resources will send a Job eXecution Time-out (JXT) message to the client. It will also remove the job from its Jobs Committed To list. A client receiving a JXT message would be forced to re-submit the original RFS message.

Pending a Job

As described above, when the designated server determines that some other local server is better able to handle the job request, it places the job object on its Jobs Pending list. The designated server also degrades the metric information for the local server. It must do this so that the comparison algorithm will not falsely determine that an unresponsive local server is best able to service subsequent jobs. Degrading usually consists of simply setting the values of metrics for the local server to zero.

If a local server is fully functional and does subsequently accept the pended job, it will broadcast a SMA message, which will be received by the other servers. The designated server will then update the metric information for the local server and remove the job from its Jobs Pending list.

Periodic Walk of Lists

To complete the algorithm, process timed-out jobs, and handle pended jobs for unresponsive local servers, the rate.d process on the designated server must periodically wake up and walk the three job lists described above.

When a server finds a committed-to job that has timed out, it drops the job from its list and sends a JXT message to the client as described above.

When a server finds a pending job that has timed out, it must re-run the bestness algorithm to see if it is capable of servicing this job. The implication is that a job the designated server thought was better handled by a non-local server was never accepted by any local server. This could be due to some catastrophic failure of the local server or any other reason causing the local server to be or appear to be unresponsive. Should the designated server now determine that it is able to accept the job, it commits to the job in the manner described in Initial Contact above. In keeping with rate.d's stated goals of high robustness, such jobs should not be thrown away. This fallback mechanism provides for jobs which were never accepted. By this mechanism, no job will ever be refused execution unless all the AGs are down. When this is the case, or if network connectivity is unavailable between the client and the AGS, the client will timeout after a reasonable amount of time.

Cluster Initialization

When a server has completed its initialization and is ready to begin accepting jobs, it must first broadcast an SMA message to the local area network. The SMA message will contain an empty job identification parameter, but will contain initial metric information for the designated server. This message serves to inform local servers of the new server's existence and insures that other servers rate it appropriately for volunteer evaluation. Optionally, local servers may respond to such initial SMA messages by relaying their own metrics to the new server via an SMA message sent point-to-point, allowing the new server to be properly configured regarding peers. In the absence of such a response, new servers become synchronized quickly as jobs are accepted by other servers and the associated SMA messages are received.

Jobs Accepted by Multiple Servers

Under certain circumstances (most notably, during cluster initialization), rate.d metrics on all the servers may not be correctly synchronized. In this case, more than one server could respond to any particular RFS message. The rate.d algorithm attempts to minimize multiple responses sent back to the client, in order to avoid response storms. However, at times two or more servers will respond with JXC messages. In this case, the client may arbitrarily choose the server from those which respond. On the non-selected servers, the committed jobs will time-out when no RFE is received. Synchronization between the servers will soon be re-established via the SMA messages, thereby eliminating future duplicate JXC messages.

The Compulsory Volunteer Algorithm

The determination of which server is best able to service a new request is called the Compulsory Volunteer Algorithm. The actual algorithm used is considered an implementation detail, although it is important to note that the same algorithm must be implemented on all servers in a cluster. The Compulsory Volunteer Algorithm can be tuned for any important criteria, such as balanced job distribution or balanced ratio of resource allocation. The need to balance the algorithm used with the service provided by the server cluster ensures that this algorithm cannot be fully generalized, though a selection of "stock" algorithms could be made available. The even job distribution algorithm is described below.

Even job distribution can use the sorting described below to determine which server should accept the job. Two metrics are required for this algorithm: the active job count and high water mark. The active job count is the number of jobs currently being serviced by this AG and high water mark is a total number of jobs this AG may process. Rate.d has no prior knowledge about the expected duration of a job.

1. Sort first on the ratio of active job counts to high water marks.
2. Sort next on some arbitrary piece of datum, such as the server's host number.

Once the sort has been completed, the server at the top of the list is determined to be the "best" for the incoming job. If this is the designated server, the job is immediately accepted. If this is a local server, the job is pended as described above. A cluster could be tuned for equal headroom ratio or any other metric determined to be appropriate for the application. Headroom ratio is the relative measure between the active job count and the high water mark.

Metrics for the Example

The first of these metrics is the active job count, describes the total number of running and committed to jobs. In other words, the active job count is a simple metric describing the amount of resources allocated for currently (or soon to be) executing service requests.

The second of these metrics is the high water mark, which describes the maximum number of jobs that a server can execute. For each class of jobs, this metric can incorporate such values as an administrator imposed ceiling, and averaging over some time period.

Integration with Existing Services

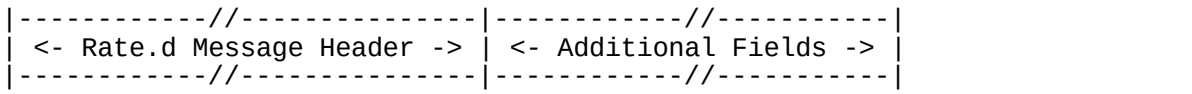
The rate.d mechanism requires that clients are able to handle service allocation separately from service provision. Many common services deployed today do not make allocation among equivalent providers a distinct element of the protocol, so changes will be required in clients to deploy the rate.d system. The changes required include making the rate.d RFS request and processing replies. If the application protocol is rate.d-aware or is extensible, the authorization provided by the rate.d daemon simply needs to be inserted into the RFE sent to the Application Protocol Daemon. Protocols designed for use with rate.d will provide fields for this, requiring, at a minimum, the job number with which the service was allocated by rate.d, and possibly an authorization ticket which may optionally be returned by rate.d. Extensible protocols, such as HTTP, may be used with added fields transmitted in header information. Protocols which use RFC822 headers or similar mechanisms to identify information passed in the request are particularly open to such extensions.

Authorization tickets (CAT) may be optionally specified by the rate.d server. Clients requesting service provision can be forced to prove their validity by providing the ticket given to them by rate.d during the service request phase. Authorization tickets prevent clients from skipping the request for service phase.

Protocols which cannot be extended at the server for various reasons may be handled in a slightly different way by providing an “interposer" between the client and the actual server. The client would be required to transmit the job number and any authorization tickets required ahead of the actual protocol data. The interposer would process this prepended information, performing any required authorization checks and sending the allocating rate.d the necessary JXS message. Once this is done, the interposer need only relay transmissions between the client and service to allow the service process and client to communication with a high level of transparency. When the communication between the client and the service terminates, the interposer must transmit the expected JXE message to the rate.d daemon. An interposer may operate in a way similar to the current inetd daemon.

Rate.d Message Details

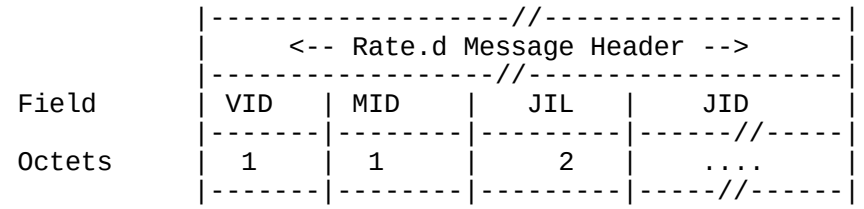
Each message starts with a message header common to all messages used to communicate among System Objects. The message header is followed by additional fields specific to each message. Some fields may include information which is specific to the application being served or encapsulates authentication or authorization tickets generated by some external mechanism.



Some messages, particularly the RFE message, may be subsumed into an application protocol which provides for all required information to be transmitted as part of the application data. This is particularly relevant for facilitating the use of rate.d with protocols designed for extensibility such as HTTP. The message formats specified here are specified to define the content of the logical messages and indicate the minimum content required to support the rate.d resource allocation system.

The Rate.d Message Header

The message header allows the receiver to identify the service request with which the message is associated, the type of the message, and the version of the rate.d protocol used by the sender.



VID

Version ID: The version of the rate.d protocol being used. This document describes version 1.

MID

Message ID: The type of message. The following values are defined for this octet:

1.RFS -- Request For Service
2.RFE -- Request For Execution
3.JXC -- Job eXecution Committed
4.JXT -- Job eXecution Time-out
5.JXS -- Job eXecution Started
6.JXE -- Job eXecution Ended
7.SMA -- System Metrics Analysis

JIL

Job ID Length: The number of octets used to encode the job identification. This may be zero.

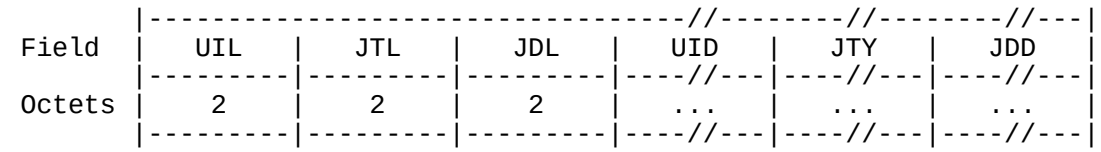
JID

Job ID: The job identification. There must be exactly as many octets as specified by the JIL field.

Rate.d Message Specific Fields

Each section below describes the format for messages of particular types. These descriptions do not repeat information contained in the message header, but describe exactly those fields which follow the message header for each message type. If no fields follow the header for a particular message type, that is indicated below.

Request For Service



Field	UIL	JTL	JDL	UID	JTY	JDD
Octets	2	2	2	...	...	...

UIL

User Identification Length: Number of octets being supplied to identify the user. This may be zero.

JTL

Job Type Length: Number of octets being supplied which specify the type of job being submitted. This may be zero.

JDL

Job Description Length: Number of octets being supplied which provide advisory information regarding the job. This may be zero.

UID

User Identification: Client identification. There must be exactly as many octets as specified by the UAL field. The format and interpretation of this field is outside the scope of this specification.

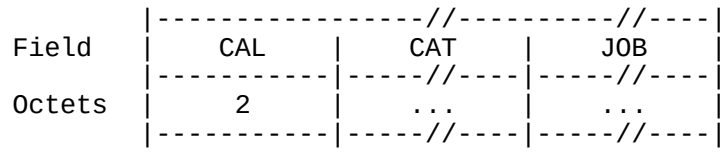
JTY

Job Type: Job type specification. There must be exactly as many octets as specified by the JTL field. The format and interpretation of this field is outside the scope of this specification.

JDD

Job Description Data: Advisory information describing the job. There must be exactly as many octets as specified by the JDL field. The format and interpretation of this field is outside the scope of this specification.

Request For Execution

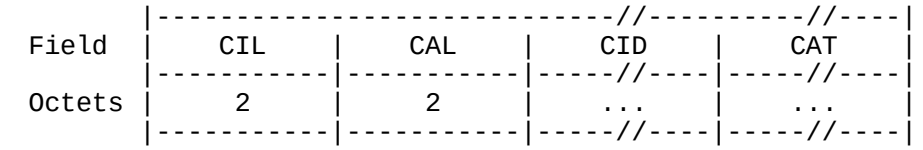


CAL
: Client Authorization Length: Number of octets being provided to the Application Protocol Daemon as an authorization ticket. This must match the CAL field of the JXC message received by the client. This may be zero.

CAT
: Client Authorization Ticket: An authorization ticket supplied by the rate.d process. This must match the CAT field of the JXC message received by the client.

JOB
: This is the data for the job. All communications between the client and the Application Protocol Daemon are entirely dependent on the application starting with this field. The length of this field is not specified and may not be known. The field may be empty.

Job eXecution Committed-to



CIL
: Contact Information Length: Number of octets being provided with encoded contact information. This may be zero.

CAL
: Client Authorization Length: Number of octets being provided to the client as an authorization ticket to provide the application on contact. This may be zero.

CID
: Contact Information Data: Application-specific information describing how the client should contact the Application Protocol Daemon. There must be exactly as many octets as specified by the CIL field.

CAT
: Client Authorization Ticket: An authorization ticket which must be supplied to the Application Protocol Daemon in the RFE message unchanged. There must be exactly as many octets as specified by the CAT field.

Job eXecution Timed-out

The JXT message requires no additional fields. (The specific job is detailed in the Rate.d Message Header).

Job eXecution Started

The JXS message requires no additional fields. (The specific job is detailed in the Rate.d Message Header).

Job eXecution Ended

The JXE message requires no additional fields. (The specific job is detailed in the Rate.d Message Header).

System Metrics Analysis

This message may be sent by the rate.d process to the rate.d processes on other servers in the cluster.

Field	NSM	SMI	SML	SMD	...
Octets	2	2	2	...	...

NSM
: Number of System Metrics: The number of system metrics for which results are being supplied with this message. This numbers is the number of triples, each consisting of one SMI, one SML, and one SMD field. The value of this field is unsigned and may be zero.

SMI
: System Metric Identifier: Value indicating what metric generated the following SML and SMD fields.

SML
: System Metric Length: Number of octets of encoded data which describe the value of the metric. This may be zero.

SMD
: System Metric Data: Metric data. The format of this data is entirely dependent on the implementations of rater.d and rate.d. There must be exactly as many octets as specified by the SML field.

Metrics Sufficient to Implement rate.d

The even job distribution example of rate.d described above made use of two metrics in applying the compulsory volunteer algorithm. In this section, we argue that these metrics, the “active job count" and “high water mark," are sufficient to implement the rate.d resource allocation system in all cases, given a way to determine the high water mark such that the requirements discussed below are met. Such an implementation shows that this system is general enough to cover all design requirements for rate.d. The definition of the active job count given in the example is sufficient and applies to this discussion.

The high water mark must be computed such that the availability of system resources is measured in load units. Load units are independent of the number of actual jobs that the system can handle, and can be tuned to the types of jobs accepted by a server cluster. This tuning may be done to favor CPU-bound processes, I/O-bound processes, or any other criteria considered critical for a cluster. Since cluster tuning is highly dependent on local or application-specific requirements, it will not be discussed in further detail here. More importantly, load units are used to place maximum load limits on each server based on the capabilities of the host, and serve to rate each host on a relative scale. These maximum limits are derived from benchmarking each server as part of the cluster configuration. Note that no server need access to this information for any other server, and only the rater.d process needs to know the actual limit for the designated server.

The maximum limit of a server's capacity expressed in load units represents an unavailable state for the server, or A0, and an unloaded system (carrying zero load units) has maximum availability, or Amax. At time t, a server will have some measurable load, Lt. This load may be expressed as some percentage of Amax. The availability of the system at any point in time may be expressed as:

$$A_t = (100\% - L_t) \times A_{max}$$

The correspondence, Ct, is the relation between jobs in the system as measured by the active job count and load units. Correspondence is defined by the equation:

$$C_t = L_t \times A_{max} / AJC$$

The high water mark can be expressed by the equation::

$$HWM = AJC / L_t = A_{max} / C_t.$$

Open Issues

Deadlocking: During prototyping we found that SMAs for local servers have to degraded over time, otherwise deadlocks can occur. Simply degrading local SMAs to zero when that server hasn't been heard from, in a given time, was found to work fairly well.

Failure to accept a job: It's possible for a pended job to never be accepted by any server. This can happen if the best server crashes, and none of the other servers can accept the job. This will only happen if the cluster as a whole is severely overloaded. The administrator will probably be running some kind of monitor on the cluster and be able to determine the existence and potential duration of such situations, in which case new servers can be added to the cluster. One preliminary implementation of an earlier version of the protocol involved such a monitor.

Metathesaurus Integration as an Example Client and Server

An early prototype of a UMLS Knowledge Sources Metathesaurus Browser was implemented to demonstrate AGS capabilities both on the server and client side.

This work is divided into three parts: 1) the Metathesaurus Data Modeler, 2) the Metathesaurus data BSD Btree Server (an example AGS service) and 3) Tool The GrailMeta Metathesaurus browser Grail applet (an example AGS client). Each of these is described in more detail below.

Metathesaurus Data Modeler and Loader Tool

In order to demonstrate the access to a popular service, the Metathesaurus information was made available via a network server. This information must be parsed from ISO9660 CD ROMs and organized into a persistent store that makes data accesses reasonably fast. For the server BSD Btrees, a freely available BTree implementation, were used. The Modeler part of the tool allows the administrator to graphically build a database schema using a schema editor. Once the schema is complete, the user can then save the schema to disk for later edits. The Loader aspect of the tool allows the administrator to load the BSD BTrees from the ISO 9660 CD ROM UMLS Metathesaurus flat files according to the model previously defined. The BTrees can be either NEXT IXBTrees or BSD BTrees. This tool is currently being used to create and manage the data sets used for the Metathesaurus BTree Server.

Metathesaurus BTree Server

The Metathesaurus BTree Server is a network daemon that serves Metathesaurus information build from the Modeler/Loader tool. The implementation is a specialized HTTP server that directly serves the contents of BTree database files (instead of HTML) based on the contents of the 'querystring' part of the URL request it receives. The server caches previous requests and can serve multiple requests simultaneously. The server is written in Python and implements the HTTP GET request. Error codes are also returned with the appropriate HTTP failure. (e.g. 403 not found). This server allows the 300-400 MBytes of data to reside on a central server somewhere (currently newcnri) and various clients can access from anywhere so long as they submit proper HTTP. For example, major portions of this server have been tested using a standard web browser such as netscape. The format of the HTTP request is as follows:

http://host:port/?database=dbname&key=keystring&command=commandname

Valid database(s) are:

mrcxt9 - Concept Contexts table indexed on Hierarchical Code
mrcxt0 - Concept Contexts table indexed on Concept Unique Identifier
mrdef0 - Definitions table indexed on Concept Unique Identifier
mratx0 - Associated Expressions table indexed on Concept Unique Identifier
mrcon0 - Concept Names table indexed on Concept Unique Identifier
mrxnw.eng1 - Normalized Word Index indexed on Normalized Word

Valid command(s) are:

lookup - Return all matches that match key
first - Return only the first match that matches key
batch - Return only the first match that matches a list of keys
unique - Return only unique matches that match key
mesh - Return a mesh context record corresponding to key
meshlevel - return a list of mrcxt records that correspond to all of the categories of a particular HCD level
meshtop - return the context records corresponding to the top of the mesh tree

GrailMeta Metathesaurus Browser Applet

GrailMeta is an applet written for Grail that allows browsing of terms in the UMLS Metathesaurus. More attention was given to the user interface aspects of this tool than the particulars of specific optimal organization of the Metathesaurus information.

One types a word to search for in the Search term entry field. By clicking Search, a list of concepts that were found to match are listed in the Matching Concepts listbox. Clicking directly on a matching concept will display that concepts definition and context(s) in the Concept Definition and Concept Contexts lists respectively. Just above the Search term entry field, there is a menu called Metathesaurus, from which one may select the Hierarchy Browser. With the hierarchy browser window visible, clicking on a concept context will display the context hierarchy in the hierarchy browser window. This is illustrated below.

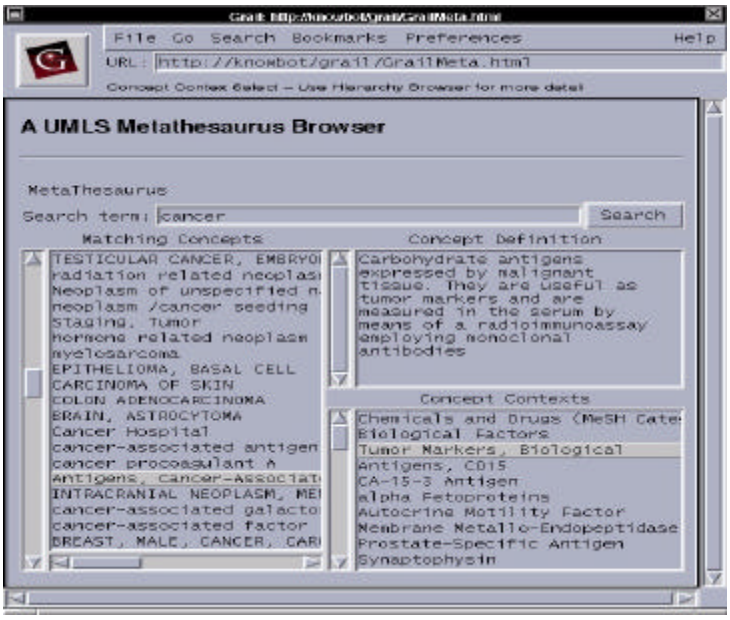


Figure 2 - The GrailMeta Grail applet.

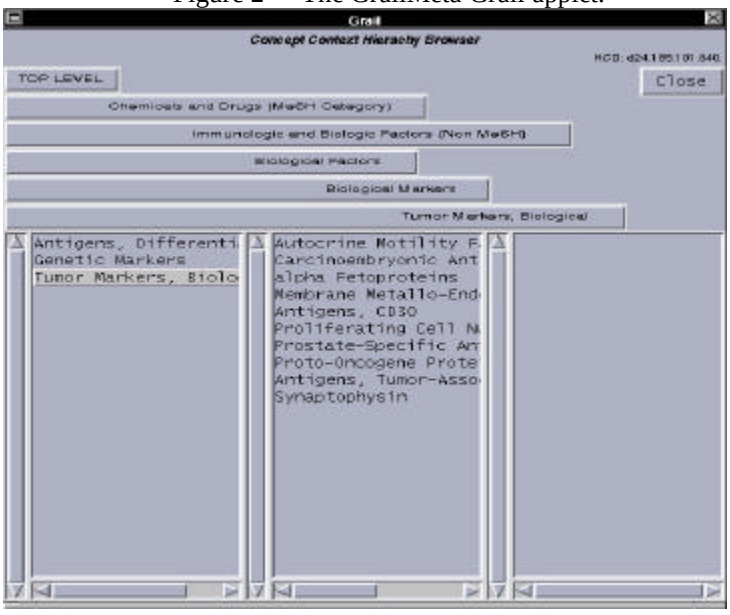


Figure 3 - GrailMeta Concept Context Hierarchy Browser.

References

1. Crocker, David H. Standard for the Format Of ARPA Internet Text Messages (RFC822).
2. Veizades, John; Guttman, Erik; Perkins, Charles; Kaplan, Scott. Service Location Protocol. IETF Internet Draft.
3. Stein, Greg; Python Database API; ftp://ftp.python.org/pub/www.python.org/sigs/db-sig/DatabaseAPI.html