Resource Allocation and Tracking System (RATS) Deployment on the Cray X1E, XT3 and XD1 Platforms

Robert M. Whitten Jr., ORNL, Tom Barron, ORNL, David L. Hulse, Computer Associates International, Inc., Phillip E. Pfeiffer, East Tennessee State University, and Stephen L. Scott, ORNL

ABSTRACT: The Resource Allocation and Tracking System (RATS) is a suite of software components designed to provide resource management capabilities for high performance computing environments. RATS was initially developed as a joint effort between Oak Ridge National Laboratory and East Tennessee State University, as part of the SciDAC Scalable Systems Software Project (SSS), with support from the National Center for Computational Sciences.-

RATS provides flexible support for various types of input data through the use of an adapter layer that supports foreign-to-virtual attribute conversion. RATS also supports the use of plug-in Python functions for data validation, and an architecture that decouples data collection from data processing. RATS currently supports resource allocation management on the Cray X1E, XT3 and XD1 platforms at Oak Ridge National Laboratory. Future directions include improved support for web access, improved support for configurable host management, and integration with LDAP, DCE, and RSA.

KEYWORDS: XT3, X1E, Cray, RATS, resource management, allocation

1 Introduction[1]

1.1 RATS as a Well-Designed Tool for Account Management

Accountability to program sponsors and funding agencies requires that large research computing centers document how their machine cycles are used. The system described in this paper, the Oak Ridge National Laboratory (ORNL) Resource Allocation and Tracking System (RATS), addresses a need for a database application that collects computer usage data from a heterogeneous set of computing platforms; provides administrators with report-based and web-based access data on system utilization; and supports the use of this data for admission control, based on a system of accounts for resource utilization. The initial phase of RATS system development, which was completed in May 2004, began in January 2003, after it was found that no single public domain accounting package met the needs that the National Center for Computational Sciences (NCCS) at ORNL confronted. The NCCS continues to use RATS as a primary tool for system and user management. The original system's architecture has proved flexible enough to provide a scalable solution for the needs of ORNL users and staff. RATS's use of available open source database management systems and integration with the installed base of batch schedulers has aided in its acceptance.

RATS accepts input in the form of usage log files from a heterogeneous set of job management systems. Pluggable input modules parse, validate, and format the data into a unified internal format that can be stored in a relational

1 Research supported by the Mathematics, Information and Computational Sciences Office, Office of Advanced Scientific Computing Research, Office of Science, U. S. Department of Energy, under contract No.DE-AC05-00OR22725 with UT-Battelle, LLC.

database. This need to support heterogeneous platforms was originally driven by NCCS's use of LoadLeveler and the Portable Batch System (PBS) on different research supercomputers. The NCCS needed to consolidate the information from host accounting files in a unified format that would allow the information to be analyzed for usage trends and subsequently be used to modify scheduler behavior. RATS provided a framework for accomplishing these goals.

The RATS data model supports association of users, projects, jobs, and allocations. One or more users are associated with one or more projects. Each project has one or more resource allocations, and one or more users that are designated as primary investigator (PI) or co-PI. This data model captures the flow of allocation usage from initial allocation to usage of allocation by user consumption through job execution.

RATS supports arbitrary suspension of operation at either front-end data gathering components or back-end data storage components. This feature is intended to allow system maintenance with minimal disruption. Thus, if a new input module is needed, the front-end can be suspended without affecting database operations. Conversely, if the database needs to be restructured, removing the database from operation for a period of time will not impede the continued collection of incoming usage records. This data, which is stored in a holding table between the RATS front and back ends, would simply be held until the database becomes available again.

Ongoing work has included support for an operational model in which formal allocations of computer resource are designated to particular projects at the beginning of the fiscal year and then usage through the year is tracked against these allocations. One mandate for RATS was to provide the machinery to carry out this allocation and tracking task.

Best practices used during the ongoing development of RATS are unit testing and code reviews. These techniques have aided in the discovery and correction of errors in both the design and implementation of RATS. These techniques aided in discovering shortcomings in the LoadLeveler API as well as variances in the PBS implementation on the Cray XT3.

It was discovered that there was no way to retrieve processor counts from the LoadLeveler API. This necessitated a redesign of the data retrieval component for LoadLeveler. The PBS implementation on the Cray XT3 uses a different attribute for processor count than on other PBS installations. This variance was handled by a small adjustment to the implementation of the PBS retrieval component. The design allowed for quick resolutions of these issues with minimal effort.

1.2 Relationship to Cray

Since RATS implementation began, the NCCS has moved away from IBM systems, replacing them with Cray systems using PBS to manage their job stream. Thus, the need to support LoadLeveler has diminished. The design decision to make RATS data collection modules pluggable to ease adding or removing support for particular batch systems was essential. Rather than losing a data source, it is conceivable that several might have been added. New data sources may yet be added in the future. The design selected for RATS has served well by providing appropriate flexibility to meet the evolving demands of the NCCS.

Due to the design of RATS, the NCCS's XT3, X1E, and XD1 systems have been successfully integrated into the system.

1.3 What Follows

The remainder of this report summarizes the history of the RATS project (§2); discusses the architecture and design of RATS (§3); describes the system's implementation (§4) and limitations (§5); and discusses plans for its future development (§6).

2 History

The project to develop a Resource Allocation and Tracking System (RATS) began early in January 2003, as a capstone project for a team of students from East Tennessee State University (ETSU) in Johnson City. In January, 2003, Rebecca Fahey and Tom Barron of the NCCS staff and Stephen Scott, a senior research scientist in the Computer Science and Mathematics Division (CSMD) at Oak Ridge National Laboratory (ORNL) visited ETSU to interview students to work on the RATS project. In this joint NCCS/CSMD effort, funding was

provided through May 2004 for the students by NCCS and for their faculty advisor by CSMD.

From NCCS's operational point of view, the effort addressed several goals:

- NCCS needed to consolidate usage data from IBM computers running LoadLeveler and Cray and SGI computers running PBS for usage tracking purposes. To the extent practical, a requirement was to decouple the input mechanism from the format of the backend database.
- NCCS needed a mechanism for allocating resources to specific projects at the beginning of the fiscal year and than tracking usage against these allocations.
- NCCS sought to develop and enhance collaborative relationships with the research and academic communities.

The project represented collaboration between the NCCS and CSMD's SSS team. From the research perspective, the project addressed three primary goals.

- Produce a platform for future research and development of center-scale resource management tools.
- Explore the feasibility of XML-based input mechanisms for such systems.
- Explore the practicality of research and production team collaborations.

ETSU approached RATS with two major goals:

- Train four ETSU students in the art of software engineering.
- Develop and enhance the opportunity for collaborative projects with entities at ORNL.

The work on RATS consumed about 8500 person-hours, divided between 7280 student hours spread over three semesters and 1200 hours for the team's advisor, Dr. Phillip E. Pfeiffer. Most of Dr. Pfeiffer's contribution was devoted to architecting RATS, serving as team liaison, and building a tool used to prototype the database (see section 4.1).

2.1 Main Challenges

Two challenges made the consolidation of batch system record formats more difficult than originally expected. The team had hoped to find open source code that would handle the resource accounting aspect of the system. Unfortunately, none of the candidates adequately addressed NCCS's accounting requirements. Thus, more effort was required to analyze, specify, and design the accounting functionality.

Secondly, a study of the LoadLeveler's API for collecting usage information showed that LoadLeveler's internal data model was hierarchical, meaning that the team had to not only map fields from one scheme to another, but also to normalize the hierarchical format data into a relational model.

Despite these challenges, the format consolidation was completed successfully. The team built a web-based interface for examining the consolidated information to ensure validity. Systematic unit tests were constructed along with the code modules, allowing the code to be validated as it was built.

2.2 Developer Status

Each of the four RATS team members found employment soon after the academic portion of the project was completed. Two were hired by ORNL, one by Computer Associates, and the fourth by an NYSE brokerage firm

3 Architecture

3.1 Architectural components

RATS is designed as a modular suite of interrelated components. The diagram in Figure 3.1 depicts the relationship between the various components in a leveled, hierarchical manner.

3.1.1 Database-specific front ends

The architecture of RATS provides for components that are responsible for interfacing with the various front-end data sources. These data sources are typically in the form of scheduler logs created by the various batch schedulers (PBS, LoadLeveler, etc.). Additional sources can be added by the creation of new components tailored to accept data in any imaginable format.

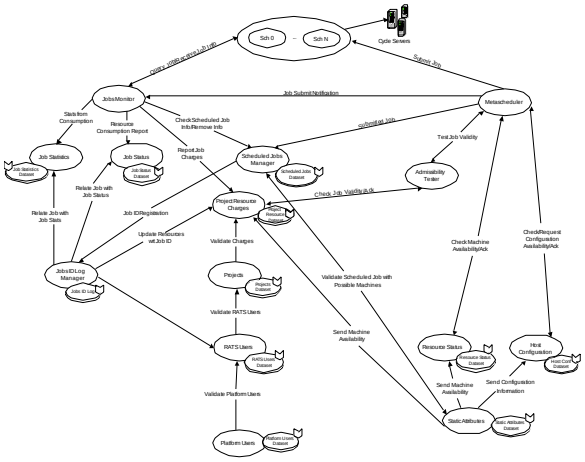

Figure 3.1

3.1.1.1 Cray

The Cray systems at ORNL use PBS as the batch scheduler. RATS has a component that parses PBS job logs to gather usage information. There are subtle differences between the PBS installations on each Cray (XT3, X1E, and XD1) with respect to field definitions (i.e., ncpus vs. size). Differences in the specific PBS installations are handled at this level with the component providing the needed data value based on the host that is being accessed.

3.1.1.2 IBM

The IBM system at ORNL uses LoadLeveler as the batch scheduler. One of the key design difficulties with RATS was the conversion of the hierarchical database LoadLeveler presents in its job log. The component that parses the LoadLeveler job logs converts the hierarchical database input into tabular output. To facilitate this conversion, an iterator design pattern was used in conjunction with LoadLeveler's API to transform data along each branch in the job log into a tuple for insertion into RATS.

3.1.2 Isolation layer

Originally, the issue of storing the data pulled from the schedulers into the main database seemed much simpler than the problem of retrieving data from the heterogeneous datasets provided by the different schedulers. However, it was quickly realized that data storage was much the same issue as data retrieval. Just as the data input sources could vary, the data output destination might also vary. The current intended destination was a MySQL database, but in the future the destination could be as simple as a flat file or as complicated as an XML stream to some web service. Just as data retrieval was abstracted away using an adapter pattern, data storage was abstracted away with "putters" that convert incoming data into a standardized tuple format. Derived classes that implement an abstract "putter" interface decide how to store incoming data, using SQL insertion operations for a database, or file system calls for a flat file, or XML documents for a web service.

This approach allowed data retrieval code to be updated and tested independently from the data storage code and vice versa—a major advantage for system development. A new iterator for a different scheduler could be coded, tested, and added without affecting the data storage code, and with minimal impact to the other iterators already present in the system. Similarly, a new data destination could be implemented without impacting data retrieval.

3.2 Data model

Figure 3.2 shows the core definitions for the RATS data model. The notation in Figure 3.2 is similar to notations in common use for defining the semantics of programming languages. The following is a rough description of how this model was used to realize the RATS schema:

- Every type definition in Figure 3.2 helped to realize one or more schemas for RATS tables. StringType, for example, realize the type of all text columns in the database.
- Names in ALL CAPITAL LETTERS were realized as built-in SQL column types and enumerations. StringType fields, for example, were realized as columns of type VARCHAR(86). Similarly, TypeOfResourceType fields were realized as MySQL enumerations with one of three values: JOB ACTION, TRANSFER ADMINISTRATIVE ACTION, or FUNDING ACTION.
- Types defined using **x** ("cross-product") and → ("map-forming") were realized as tuples. JobAttributesDatasetType, for example, was realized as a 28-tuple that captures data from a PBS job record. Definition nesting also affected tuple realization.

- Nested definitions were collapsed, where possible, to form individual tuples. For example, TransactionLogType realized a 7-tuple because *Amount* represents a pair of StringType values.
- The → operator was realized as a uniqueness constraint on the fields that precede it. For example, the RATS transaction log was realized as a table with a key field of *ID*.
- Names of types like **TransactionLogType** that realized RATS tables are highlighted in **boldface**. Other type definitions, like StringType, were created for clarity or to express relationships amongst types—i.e., "join constraints" in ER diagrams.
- Field names like *ProjectID* were realized as column names. The transaction log's columns, for example, are named *ID*, *Type*, *Source*, *Sink*, *Amount.ComputingResources*, and *Amount.Value*.
- Types defined using + ("disjoint union") were realized as one or more fields that collectively contain one of several distinct types of data. NamedComputingResourceType, for example, was realized as a column that can contain one of three types of data: the name of a CPU; the name of a disk; or the name of a main memory element.
- The notation $2^{SomeType}$ denotes a set of items of SomeType. A field *F* of type $2^{SomeType}$ was realized by defining a supporting table that paired values of SomeType with numbers that identified the sets to which they belonged. These numbers were then entered in field *F*. For example, sets of subprojects for the Project Data Table, a table of type ProjectDataType, were realized by creating a single, all-inclusive supporting table that paired subproject IDs with the IDs of the sets to which they belonged, then entering these set IDs into the project data table's *Subprojects* column.

This type-based approach for defining RATS content provided a precise, flexible, and fast basis for developing the data model. The need for agility and precision became apparent at the project's outset, after an day-long JAD session failed to produce agreement on an initial accounting model—let alone substantive requirements for the RATS database. Additional concerns about the difficulty of scheduling follow-on meetings suggested a need to present clients with simple, carefully defined models that they could critique independently.

The notation shown in Figure 3.2 met the needs described above. The notation is clear, in the sense that it presents clients with a hierarchical taxonomy of definitions: a strategy for expressing data models that seems more intuitive to clients than E-R diagrams. The notation is flexible, in the sense of being modular: definitions at one level of the taxonomy can typically be changed without affecting dependent definitions. Finally, the notation is precise, in the sense that straightforward procedures can be defined for transforming these type definitions into SQL CREATE TABLE statements.

4 Implementation

RATS is implemented using a variety of technologies. The job monitor component is composed of a series of information retrieval modules in conjunction with an equivalent set of storage modules. An abstraction layer that defines the various attributes that RATS will manage separates the retrieval and storage modules. These modules are implemented as C++ classes with three total classes per attribute. One class each represents retrieval, storage, and abstraction. Modules currently exist for accessing PBS, LoadLeveler, and text file data sources. MySQL and text file modules exist for storing normalized data values for subsequent querying and processing. Data validation modules exist in the form of Python scripts, which act as filters between data retrieval modules and abstraction modules to provide flexible value checking of raw data inputs before storage by the data storage modules. These validation modules use regular expression parsing to validate data based on predefined criteria. The validation schema also provides for disabling of all validation in cases of performance issues or testing.

RATS currently uses MySQL for all data collection and reporting. One reason for choosing MySQL was its feature set. MySQL provides all the standard kinds of functionality one expects from a database, including transaction processing, simple SQL queries, and report generation. Others included MySQL's speed, simplicity, and a desire to avoid the use of special features in more complex databases, as a way of keeping NCCS's options open if the need to switch databases would ever arise. There was

nothing esoteric about how the database was used—some other standard RDBMS like Oracle or Sybase could be used in place of MySQL.

4.1 Realizing the Data Model

The concerns about the difficulty of establishing requirements, coupled with concerns about the need to accommodate last-minute changes to the model, led to the development of the markup-language-based data modeling language depicted in Figures 4.1.1 and 4.1.2, and a compiler for transforming a model rendered in this language, Model-T, into an SQL program for creating a database's tables. The final Model-T definition of the RATS data model, which was produced in Spring 2004, consisted of 116 definitions, 35 of which were compiled into tables.

The Model-T compiler allowed the RATS team to implement last-minute changes to the data model quickly, as the vision for what the database should do crystallized. A typical Spring 2004 review of the model yielded a set of changes that could be implemented in 20 minutes, including 3-5 minutes worth of time to compile a new schema. The clients quickly become comfortable enough with the notation to review the Model-T characterization of the database directly.

The approach's primary drawback was the time needed to develop the Model-T compiler. The effort, a one-person project, took about 8 months' worth of steady work. The first two months of this effort were used to master XML, DTDs, SAX, and the compiler's implementation language, Python. A third month was spent on the development of a rule-based test harness, ADEPT, that used Python's eval() operator to eliminate the need for procedural test code [ADEPT]. An initial, 12,000 line version of the Model-T compiler was developed during the next four months of the project, together with a 12,000-case test suite. The final month was devoted to cleanup, including a metaprogramming-based rewrite of Model-T that cut the program's size and supporting test suite in half. The development work included support for definition reordering and various types of error management: e.g., checks for improperly formed SQL expressions, and the ability to continue compilation in the presence of malformed definitions.

The work on Model-T also created friction between the clients, who felt that the developer's time would be better spent on more immediate concerns, and the developer, who saw the tool as a hedge against last-minute changes in requirements. Both perspectives, in retrospect, had merit: the compiler took far longer to develop than anticipated, but proved useful for rapid database development at a critical juncture in the project's life cycle.

4.2 Data validation

RATS supports a flexible mechanism for data validation: one that allows checks on incoming data to be updated dynamically without stopping the system. This flexibility was achieved with a supporting, stand-alone (Python) application that accepts an attribute, validates the attribute's value against a pluggable validation rule, and then returns a response. These validation rules can be framed in terms of regular expression matching as well as explicit matching. Validation can also be a pass-through operation where no actual validation occurs. Pass-through validation has proven useful for testing and diagnostic purposes.

Any data that cannot be validated is placed in a suspense file for later examination by an administrator. Currently, if a single attribute fails validation, the entire record is considered suspect and is excluded from the production database. It is possible to configure the validation to accept suspect records into the production database. However, it was determined that allowing this practice might lead to an abundance of corrupt data and is therefore disabled.

The validation is performed as each new record is being processed by RATS. The validation rules are organized as a mapping between attribute names to lambda expression (Figure 4.2). This allows for flexible definitions on a per attribute basis. The validation scheme can also be changed dynamically by modifying the rules and then reinitializing the validation process.

In practice, the need for validation of this kind is real. Errors from the schedulers have been detected by the validation process. However, further refinements are still needed. For instance, verifying that a project identifier exists in not sufficient. What is needed is verification that a project exists and that a particular user has authorization to charge against that project. This

additional level of checking is beyond the scope of individual attribute validation but can be implemented using the validation design present in RATS. Validating relationships involving two or more attributes would also be useful: an example of this would be verifying that a project is valid for a given host and user.

4.3 Isolation layer

The isolation layer allows for the separation of data gathering and data query. The isolation layer consists of a data holding area that stores RATS data awaiting final movement to the long term data store. This architecture allows for maintenance or development on the main data store while data gathering continues from the schedulers or other data sources. This further allows data to be queried from the long term data store while data gathering from data sources is suspended for maintenance or development.

The isolation layer is implemented using database tables as the long term storage and holding area. Data input from the data gathering components are stored in the holding area table. A transfer utility periodically moves data from the holding area to long term storage. During this transfer, data is stored in multiple tables that separate job statistics from executable statistics. If it is deemed useful at a later date, greater separation of the gathered data could be achieved by refining the transfer procedures.

This approach is similar in design to the asynchronous publish-subscribe pattern. The data gathering components can be thought of as the publisher. The database can be thought of as the sole subscriber. One divergence from the asynchronous publish-subscribe is that the database itself is not responsible for pulling data; instead all data is pushed to the database system.

The isolation layer is moderately useful in production. Currently, data is read with a resolution of a day. The need to suspend data gathering for more than a day has not been required. However, should the system be required to read data in a closer to real time manner, the ability to suspend parts of the data acquisition while allowing operation to continue will be essential.

4.4 Data Access

Data access is accomplished through the use of various scripts which are written in Perl, Python, PHP, or C++. The use of MySQL as the database back-end has allowed scripts to be written in any of these languages with little knowledge needed of RATS internals by the individual script writer. This allows various developers, system administrators, and support staff to develop scripts to access RATS data that are tailored to their individual needs.

4.5 Unit testing

Unit testing was conducted on all classes and components initially developed. The primary unit testing framework was CppUnit. This provided a common framework and ensures a consistent regimen. The development regimen consisted of implementing a test class for each class developed. This increase in workload yielded a system that has seen little refactoring to correct errors. Most refactoring has been to add features or correct design flaws as opposed to syntax or semantic errors.

One difficulty with providing access with different technologies (PHP, Perl, etc) is that unit testing becomes more difficult and time consuming, in part due to the difficulty of getting multiple developers to unit-test their custom scripts. Code reviews and stricter configuration management than are currently in place will eventually be needed to maintain the quality of the RATS code base.

5 Limitations

The RATS project was tasked to devise a system that integrated information from three kinds of data sources: a relational database, PBS; a hierarchical database, LoadLeveler; and a set of XML documents defined by the SciDAC SSS standards for wire-level data transfer [Jackson]. The team achieved the goal of integrating data from PBS and LoadLeveler, but achieved only limited progress towards the third.

Data from XML documents could be loaded into the current RATS database by creating an adaptor, similar to the LoadLeveler adaptor, that would (1) use an iterator to enumerate every path from the original document's root to one of its leaf nodes; (2) recast this series of elements as an ordered tuple; then (3) store the tuple's component subtuples in the appropriate RATS

tables, according to a strategy that would be determined from the document's format. The relative positions of node siblings could also be captured by an iterator, if needed, and stored in a specially designated field.

Unfortunately, the task of recovering an arbitrary XML document from an RDBMS is a more difficult problem to manage. This task could be accomplished by first using joins to generate tuples that correspond to the root-to-leaf-node paths from the original document, then fusing the tuples. This code, however, would be harder to write, due to the use of joins and uncertainties about the size of the original document.

The problem of developing support for the SSS protocols, moreover, was complicated by two additional concerns. The first was a requirement for processing SSS messages that represented XQuery-like requests for data in the RATS database: a problem that would have required the implementation of a XML query processing engine on top of RATS, in addition to document reconstitution. The second was the relative immaturity of the SSS protocols at this time, which used internally developed, non-W3C standards for defining queries. Furthermore, because of the ongoing development of SSS protocols, they changed somewhat throughout the course of the RATS project. This instability made the development of a set of ad hoc interfaces between RATS and the SSS protocols a long-term risk for maintenance.

XQuery-based access to RATS could, in theory, be supported by extending the Model-T compiler so that it automatically generates an XQuery-to-SQL interface as it generates a schema from a set of type definitions. An alternative approach to generating XML views of RDBMSs that uses predefined queries as a basis for reconstituting documents is described in [SilkRoute].

6 Future development

Further RATS development will involve improving the user interfaces, host configuration support, and integration with system data sources such as LDAP.

6.1 Web-enabled RATS

Retrieving usage statistics and allocation reports are of vital concern to NCCS management and project Principal Investigators (PI). Increasing the accessibility of RATS via a web interface will facilitate timely management decisions. Using MySQL as the database backend assists in developing robust web applications to provide this functionality to users. Technologies such as PHP or JSP have established techniques for accessing MySQL.

6.2 Host configuration

Part of completing the feedback loop for RATS involves enabling usage trend information from RATS to affect the configuration and usage of the systems being managed. This is envisioned to include aspects like cluster configuration, fair share scheduling, accessibility, and detecting and quarantining failing nodes so they do not affect future jobs.

With appropriate feedback, RATS can provide a single point of control for managing host configuration across clusters. This is less applicable for more tightly integrated systems like most Cray machines.

Fair share scheduling involves monitoring usage patterns and adjusting resource availability to match user allocations over time. Not all scheduling systems support fair share scheduling directly, but a goal for the RATS project is to facilitate fair share scheduling based on usage data.

RATS can provide a single point of management for controlling user and project access to system resources. Since RATS can accommodate multiple systems in its database, it can manage a database of users across multiple systems, controlling access by resource.

Detecting and quarantining failing nodes can help maximize system availability by avoiding situations in which a failing node hangs a job and locks up the other nodes claimed by the job.

6.3 LDAP/DCE/RSA integration

The computing resources at NCCS utilize several different tools and technologies to provide systems access, security, and legacy support. The Lightweight Directory Access Protocol (LDAP) is used in conjunction with RSA SecurID to provide one time passwords and system access. IBM's Distributed Computing Environment (DCE) support is required for legacy systems such as the High Performance Storage System

(HPSS). Maintaining these various data sources can be error prone and consumes significant staff resources to maintain the consistency of user data.

Integrating RATS with these services will alleviate many of the consistency and validity issues that can occur. RATS would serve as the authoritative, central data source. In this role, additions and modification can be made in RATS and then this data can be distributed to all other systems via a push-pull design. Automated scripts will be used to synchronize the various data sets. This will reduce the number of updates a staff member has to make as well as provide a uniform interface.

7 Conclusion

RATS was developed to address the resource management issues presented in a high performance computing environment. Using a modular design, RATS successfully addresses the problem of multiple data sources of usage information. Using pluggable modules for validation, storage, and retrieval of usage data allows for flexibility and scalability. These features of RATS are well suited for deployment on Cray architectures.

8 References

[ADEPT] Pfeiffer, P., *Twelve Thousand Test Cases and Counting: a Critique of Lightweight Methodologies in Python Program Development*, PyCon 2004, Washington DC, 25 March 2004. *http://www.python.org/pycon/dc2004/papers/8/*

[Jackson] Jackson, S., Bode, B., Jackson, D., Walker, K., Scalable Systems Software Resource Management and Accounting Protocol (SSSRMAP) Wire Protocol, Resource Management and Accounting Notebook, http://www.scidac.org/ScalableSystems/

[SilkRoute] Fernandez, M., Kadiyska, Y., Suciu, D., Morishima, A., and Tan, W.-T., Silk route: A framework for publishing relational data in XML, ACM Transactions on Database Systems, 2002, 27(4):438-493.

About the authors

Tom Barron is a member of the HPC Operations Group of the National Center for Computational Sciences at Oak Ridge National Laboratory. David Hulse is employed by Computer Associates International, Inc. Phillip Pfeiffer, Ph.D., is a professor of computer science at East Tennessee State University, Johnson City, TN, and can be reached at phil@etsu.edu. Stephen L. Scott, Ph.D., is a Senior Research Scientist in the Network and Cluster Computing Group of the Computer Science and Mathematics Division at Oak Ridge National Laboratory. Robert Whitten Jr. is a member of the User Assistance and Outreach Group of the National Center for Computational Sciences at Oak Ridge National Laboratory and can be reached at whittenrm1@ornl.gov.

Figures

ProjectDataType: *ProjectID*: StringType → **(** *Account*: ResourceAccountType **x** *Subprojects*: **2** ^ProjectIDType **)**
ResourceAccountType:
ResourceAccountID: StringType → (*SetOfTransactions*: $2^{\text{TransactionIDType}}$ **x** *OverdraftPrivileges*: BooleanType **x** *Balances*: BalancesType **x** *IsFrozen*: BooleanType)
BalancesType:
SiteName: SiteNameType **x** *HostName*: HostNameType **x** *Resource*: ResourceNameType → *Value*: ResourceValueType
TransactionLogType:
ID: TransactionIDType → **(** *Type*:TransactionType **x** *Source*:FundingAccountIDType **x** *Sink*: FundingAccountIDType **x** *Amount*: ChargeType **)**
TransactionIDType: StringType
TransactionType: ENUM**(** JOB ACTION, TRANSFER ADMINISTRATIVE ACTION, FUNDING EXPIRATION **)**
FundingAccountIDType: StringType
ChargeType: *ComputingResources*: NamedComputingResourceType **x** *Value*: ResourceValueType
UserDataType:
UserName: UserNameType → **(** *SiteName*: SiteNameType **x** *SetOfRatsRoles*: **2** ^RatsRoleType **x** *FullName*: StringType **x** *Phone*: StringType **x** *AlternatePhone*: StringType **x** *E-mail*: StringType **)**
RatsRoleType: ENUM**(** SYS ADMIN, SYSTEM TASK, USER, RATS ADMIN, RATS DEVELOPER, RATS USER, RATS TASK **)**
CostOfResourceType:
Site: SiteNameType **x** *Host*: StringType **x** *ResourceType*: TypeOfResourceType **x** *Resource*: NamedComputingResourceType → *Value*: ResourceValueType
SiteNameType: StringType
TypeOfResourceType: ENUM**(** CPU, DISK, MAIN MEMORY **)**
NamedComputingResourceType :
(*CPU*: ResourceNameType + *Disk*: ResourceNameType + *MainMemory*: ResourceNameType **)**
ResourceNameType: StringType
ResourceValueType: FixedPointType
JobAttributesDatasetType:
JobName: StringType **x** *StepID*: StringType **x** *ErrorPath*: StringType **x** *GroupName*: StringType **x**
JobClass: StringType x OutputPath: StringType x ShellPath: StringType x AccountNumber: StringType x
StepComment: StringType x StepCompletionCode: IntType x StepCompletionDate: TimeType x
StepHostList: StringType x StepInputFile: StringType x StepMachineCount: IntType x
StepParallelMode: IntType x StepPriority: IntType x StepStartDate: TimeType x
StepWallClockLimitHard: TimeType x StepWallClockLimitSoft: TimeType x UserName: StringType x
CreateTime: TimeType x Depend: StringType x JobSubmitTime: TimeType x JobSubmitHost: StringType x
InteractiveJob: IntType **x** *ExecutionTime*: TimeType **x** *EnvironVarList*: StringType **x** *StopDispatchTime*: TimeType

StringType:	VARCHAR(86)
TimeType:	TIME
FixedPointType:	DOUBLE
IntType:	INT
BooleanType:	BOOL

Figure 3.2: RATS data model (abridged).

```
<!ENTITY % BOOLEAN   " ( True | False ) " >
<!ENTITY % TYPENAME-ATTR " type-name  CDATA  #REQUIRED ">
<!ENTITY % CODEGEN-ATTR   " suppress-codegen   %BOOLEAN;       'False'   ">
<!ENTITY % FIELDNAME-ATTR  " field-name  CDATA  #REQUIRED ">
<!ENTITY  % CODE-FILE-ATTR   " code-file-name  CDATA  #IMPLIED " >
<!ENTITY  % LOG-FILE-ATTR    " log-file-name  CDATA   #IMPLIED " >
<!ENTITY  % COL-NAME-CONTROL-ATTR  " use-field-names-as-column-names  %BOOLEAN;  #IMPLIED ">
<!ENTITY  % AUTOINC-CONTROL-ATTR  " autoincrement-numeric-keys    %BOOLEAN;   #IMPLIED " >
<!ENTITY   % DISCRIM-ATTR    " discrim-field-name  CDATA   #IMPLIED ">
<!ENTITY   % DEFAULT-VARIANT-ATTR  " default-variant-field  CDATA   #IMPLIED ">
<!ENTITY % ENUMLIST-ATTR  " enum-list  CDATA  #REQUIRED ">
<!ENTITY % ENUMSEP-ATTR   " enum-list-separator  CDATA  ' '  ">
<!ENTITY % ENUM-DEFAULT-VALUE-ATTR  " default-value   CDATA  #IMPLIED  ">
<!ENTITY % TARGET-DB-TYPE-ATTR  " as-type  CDATA  #REQUIRED ">
<!ENTITY % CONFORMING-TYPE-ATTR " value-check-expr  CDATA  'object'  ">
<!ENTITY % TARGET-DB-DEFAULT-VALUE-ATTR " default-value  CDATA  #IMPLIED  ">
<!ENTITY % TYPE-INITIAL-ONLY   " set |  map " >
<!ENTITY % TYPE-INITIAL-OR-FOLLOWER   " tuple | union " >
<!ENTITY % TYPE-FINAL    " instance-of | reference | dest-type | enum" >
<!ENTITY % TYPE-FOLLOWER    " %TYPE-INITIAL-OR-FOLLOWER; | %TYPE-FINAL; " >
<!ENTITY % TYPE-FORMER   " %TYPE-INITIAL-ONLY; | %TYPE-FOLLOWER; " >

<!—Root element for grammar -->
<!ELEMENT  dataset-specification   (  define-type  )*  >
<!ATTLIST    dataset-specification    %CODE-FILE-ATTR;  %LOG-FILE-ATTR;
                    %COL-NAME-CONTROL-ATTR;   %AUTOINC-CONTROL-ATTR;  >

<!-- subgrammar for supporting type definitions -->
<!-- type-defining elements -->
<!ELEMENT  define-type  ( %TYPE-FORMER; ) >
<!ATTLIST    define-type  %TYPENAME-ATTR; %CODEGEN-ATTR;  >
<!-- non-final type-forming elements -->
<!ELEMENT  set            ( %TYPE-FOLLOWER; ) >
<!ELEMENT  map          ( map-field, map-field+ )  >
<!ELEMENT  map-field    ( %TYPE-FOLLOWER; )  >
<!ATTLIST    map-field     %FIELDNAME-ATTR;    >
<!ELEMENT  tuple          ( tuple-field+ )        >
<!ELEMENT  tuple-field  ( %TYPE-FOLLOWER; )  >
<!ATTLIST    tuple-field    %FIELDNAME-ATTR;  >
<!ELEMENT  union          ( union-field+ )  >
<!ATTLIST    union          %DEFAULT-VARIANT-ATTR;  %DISCRIM-ATTR;  >
<!ELEMENT  union-field  ( %TYPE-FOLLOWER; )  >
<!ATTLIST    union-field   %FIELDNAME-ATTR;  >
<!-- final type-forming elements -->
<!ELEMENT   reference   EMPTY >
<!ATTLIST  reference     %TYPENAME-ATTR;  >
<!ELEMENT  instance-of  EMPTY >
<!ATTLIST  instance-of    %TYPENAME-ATTR;  >
<!ELEMENT   enum        EMPTY  >
  <!ATTLIST   enum       %ENUMLIST-ATTR;   %ENUMSEP-ATTR;   %ENUM-DEFAULT-VALUE-ATTR;  >
<!ELEMENT   dest-type   EMPTY >
  <!ATTLIST   dest-type
   %TARGET-DB-TYPE-ATTR;   %CONFORMING-TYPE-ATTR;  %TARGET-DB-DEFAULT-VALUE-ATTR; >
```

Figure 4.1.1: Grammar for the Markup-Oriented DEfinition Language with Types, a.k.a. Model-T

```
<define-type type-name="ProjectDataType">
  <map>
    <map-field field-name="ProjectID">   <instance-of type-name="ProjectIDType"/>          </map-field>
    <map-field field-name="ProjectData">
      <tuple>
        <tuple-field field-name="Account">  <instance-of type-name="ResourceAccountType"/> </tuple-field>
        <tuple-field field-name="Subprojects">  <reference type-name="SetOfProjectIDsType"/>  </tuple-field>
      </tuple>
    </map-field>
  </map>
</define-type>

<define-type type-name="SetOfProjectIDsType">
  <set>  <instance-of type-name="ProjectIDType"/>  </set>
</define-type>
```

Figure 4.1.2: Partial definition for ProjectDataType, rendered in Model-T

```
ValMap['GROUPNAME'         ] = ((lambda x: StringValidator(x) ), (lambda x: x.isAlphaNumeric() ) )
ValMap['INTERACTIVE'       ] = ((lambda x: int(x) ), (lambda x: 0 <= x <= 1 ) )
ValMap['COMPLETIONTIME'] = ((lambda x: StringValidator(x) ), (lambda x: x.isDateTime() ) )
ValMap['JOBID'             ] = ((lambda x: StringValidator(x) ), (lambda x: x.isJobID() ) )
```

Figure 4.2 Example Validation Rule Mapping