Technical Report NEESgrid-2003-11
www.neesgrid.org

(Draft Whitepaper Version: 1.0)
Last modified: June 3, 2003)

Preliminary Performance Analysis of the NEES Metadata Service (NMDS)

Joe Futrelle[1]

[1] National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign

Acknowledgment: This work was supported primarily by the George E. Brown, Jr. Network for Earthquake Engineering Simulation (NEES) Program of the National Science Foundation under Award Number CMS-0117853.

Feedback on this document should be directed to futrelle@ncsa.uiuc.edu

1 Summary

The NEES Metadata Service (NMDS) provides applications with the ability to create, manage, link, and discover metadata objects in the NEES repository. This study investigates the performance of the current development version of the NEES repository implementation and discusses how the results impact how the repository implementation should be used and further developed.

The performance of the NMDS appears from preliminary analysis to be determined largely by the indexing performance of the underlying database implementation. The study shows that the current NMDS implementation scales to order 10^7 objects. Object creation and update performance is better than $O(\lg n)$, and retrieval performance is almost $O(1)$. Latencies for most operations are within the 10ms range. This means that evaluating the scalability of the repository for many more orders of magnitude is impractical.

2 Overview

The NEES Metadata Service (NMDS) provides applications with the ability to create, manage, link, and discover metadata objects in the NEES repository. Client applications access the repository through an API that is made available to them through the web services framework and in future releases through the OGSI secure web services framework. The NEES repository is designed to scale to large numbers of objects and clients. This study investigates the performance of the current development version of the NEES repository implementation and discusses how the results impact how the repository implementation should be used and further developed.

3 Test environment

The NEES repository implementation was tested on the NEES central repository server at NCSA. The server is a 4-processor 500Mhz Dell 6250 server with a 256GB fiber channel disk and GigE networking, running Red Hat Linux 7.2 (kernel version 2.4.9-31smp). Backend storage for the test repository implementation was provided by MySQL. The repository server also provides NFS access to NCSA's mass storage system, but for performance reasons this is used for archiving of files, not metadata objects. Since metadata objects are stored in the database backend and are relatively small, this does not place practical limits on the number of metadata objects in the repository, since infrequently used objects can be written to files and archived in mass storage.

The MySQL Connector JDBC driver was used to access the backend database. The driver allows remote clients to access a MySQL service over a network. For this study, MySQL was installed locally on the machine running the tests. Likewise, the NMDS client API allows remote clients to access the metadata repository using a web services protocol. Because of the high latency introduced by the web services layer (which will be investigated in a later study), the local version of the API was used instead. This local version is used in the current implementation when XML files in the NEES metadata schema description and object interchange format ("NEESML"[1]) are parsed and used to initialize the repository. In future releases, the repository will accept files in this format and will translate them into calls in the local version of the NMDS API.

The NMDS client implementation includes in-memory caching which can prevent unnecessary calls to the NMDS service and the database. For the purposes of these tests, non-cacheable "worst-case" access patterns were used. Later studies will examine more typical cases, which will likely exhibit significant performance gains due to caching, and will investigate the performance impact of various caching strategies.

Initial tests indicate that performance of the repository is affected greatly by the performance of the indexes on the backend database. Without indexing, performance was poor, scaling roughly linearly as objects were added or updated. When appropriate indexes were added, performance increased substantially.

[1] The NEESML format will be described in a future whitepaper.

4 The tests

Several aspects of repository performance were tested. First, several different operations were tested, including creating objects, updating objects, and retrieving objects. Second, a number of scaling aspects were investigated, including performing operations on large numbers of objects and having large numbers of clients access the repository simultaneously.

4.1 Creating objects

Creating a metadata object involves two low-level operations: looking up the object representing the new object's type; and, selectively copying that object. In the test, 10^7 objects were created, all of the same type. Copying objects performs differently depending on the number of database rows used to represent the object; for the purposes of this test, a medium-sized object was used. The object representing the type was allowed to be cached, since in a typical application creating large numbers of objects, the number of types will be small and invariant even as the number of objects grows.

Performance was slightly better than O (lg n). The average number of objects created per second was 99.7. Figure 1 shows the test results.

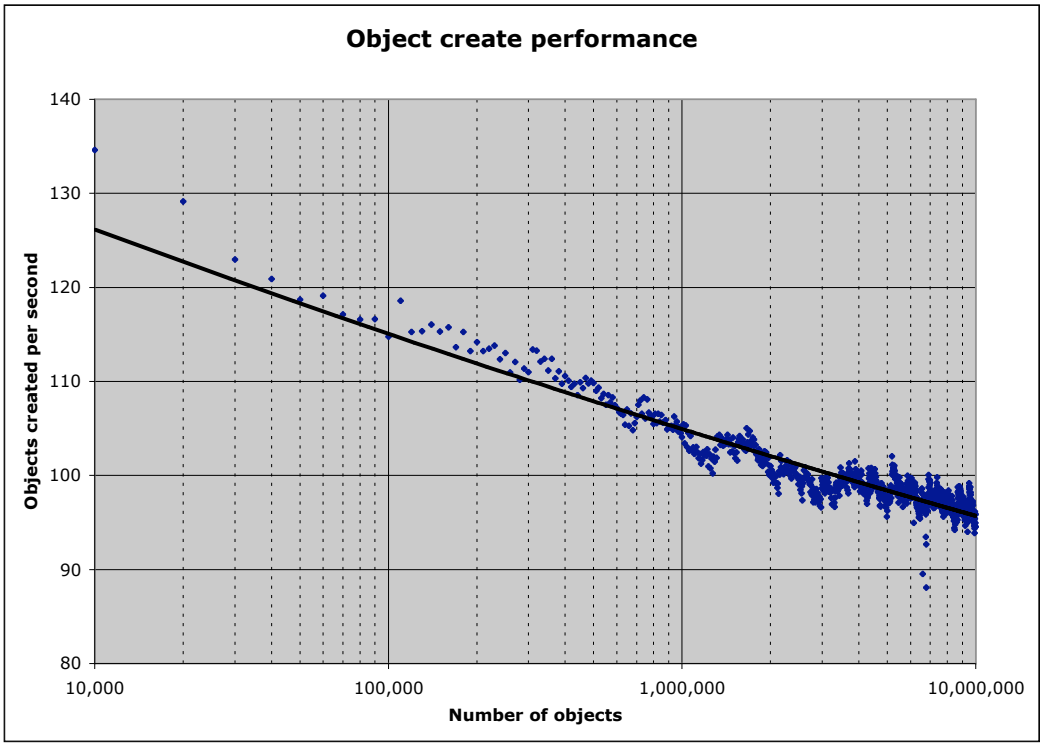

Figure 1: Object creation performance for 10^7 objects

4.2 Updating objects

Since metadata objects are immutable, updating them involves creating new versions of them. This is done by translating in-memory representations of the objects into a series of SQL statements. Unlike object creation, updating objects requires concurrency control, since a client updating an object cannot prevent other clients from obtaining a reference to the object before all the statements have been executed. The metadata service therefore only allows one update to be executed at a time. This means the per-client update time increases at order $O(n)$ in the worst case, in which all clients attempt to continuously update objects and are forced to wait in line. Figure 2 shows the performance impact of the growth in the number of new object versions, up to 10^5. Performance is comparable to, but somewhat slower than, object creation.

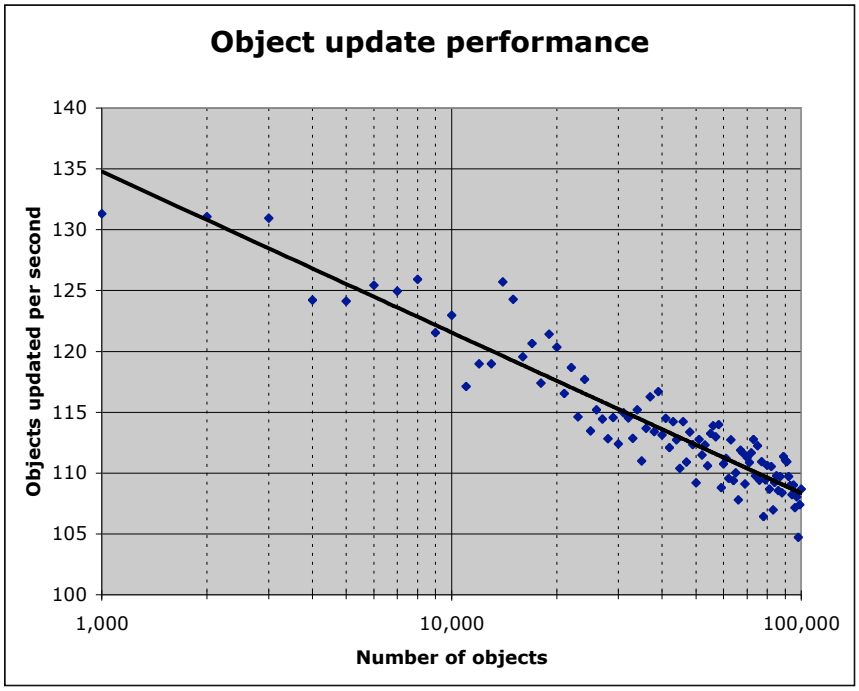

Figure 2: Object update performance for 10^5 object versions

The total number of objects updated per second also decreases in the worst case as clients are added. This is likely because of a combination of the growth in the number of new object versions and overhead incurred by thread scheduling in the Java virtual machine. Figure 3 shows the total number of objects updated per second for increasing numbers of clients. In the test, each client created an object and then attempted to continuously update it. The total number of objects created per second was estimated by multiplying the per-client update rate for the nth client by n, but this very likely underestimates update performance by failing to compensate for the effect shown in Figure 2.

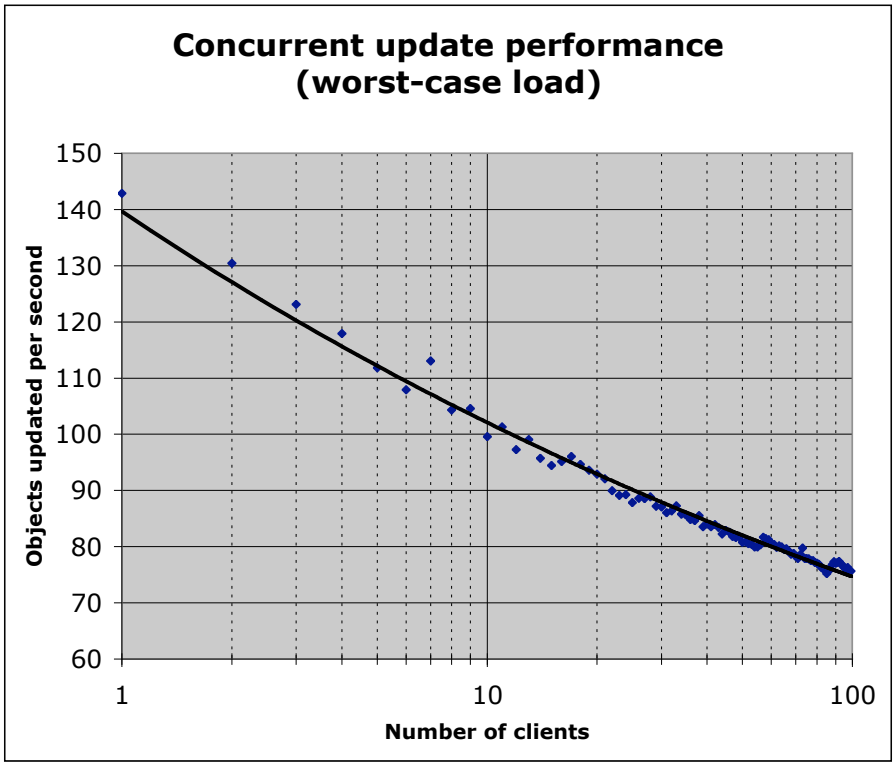

Figure 3: Concurrent update performance for 100 clients

Ad hoc tests indicate that update performance is significantly better under typical, better-than-worst-case access patterns, in which clients update groups of objects in bursts rather than continuously. This will be rigorously investigated in a later study.

4.3 Retrieving objects

Retrieving an object involves performing an SQL select statement on the object's ID and version number. For the test, medium-sized objects were created and retrieved, and only the retrieval time was measured. Retrieving recently created objects may have caused the test to overestimate performance because of optimizations internal to MySQL, but this possibility was not investigated.

Retrieval performs in almost constant time, and is roughly twice as fast as object creation. The sawtooth-shaped distribution of retrieval rates with a period of roughly 7.5×10^5 objects likely reflects MySQL's indexing performance. Since objects are immutable, no concurrency control is required for object retrieval; hence, no investigation of concurrent retrieval performance was done.

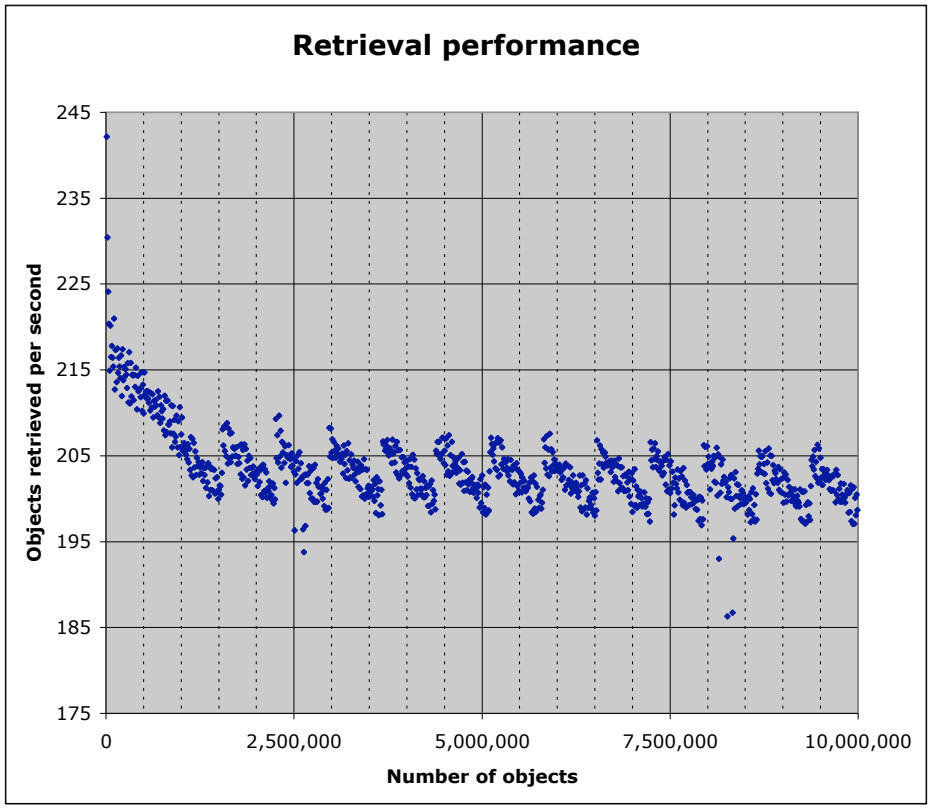

Figure 4: Retrieval performance for 10^7 objects

5 Discussion

The performance of the NMDS appears from preliminary analysis to be determined largely by the indexing performance of the underlying database implementation. When appropriate indexing commands are added to the MySQL configuration provided with the Alpha 1.1 release, the NMDS scales to order 10^7 objects. Object creation and update performance is better than O (lg n), and retrieval performance is almost O (1). Latencies for most operations are within the 10ms range. In practical terms, this means that an operation on order 10^k objects takes at least order 10^{k-7} days. Therefore, evaluating the scalability of the repository for many more orders of magnitude is impractical.

A major scaling issue is concurrency control. In the current implementation, it is assumed that all clients will access the repository through a single Java virtual machine, and concurrency control is managed using Java's threads-based system of locks and monitors. From this preliminary study, it appears that this strategy has several problems. First, the locking strategy is conservative, forcing all clients to wait whenever any object is being updated, even though some of them may be waiting to update a different object. Second, managing threads incurs overhead in the Java VM, which may significantly degrade per-client performance. And finally, where queuing cannot be avoided, clients may block for impractical amounts of time during updates. There are several possible changes that would address these issues. Most importantly, RDBMS-level transactions could be used to make operations requiring multiple SQL statements atomic. This would eliminate most

of the need for thread-based concurrency control. Also, non-blocking batch processing could be introduced so that clients needing to perform many operations could submit groups of operations and be notified asynchronously of successful or unsuccessful completion.

6 Topics for future study

Several areas require further study. The most natural extension of the current study is to duplicate the tests with orders of magnitude more objects, but this cannot be practically accomplished for more than order 10^8 objects because the tests would take too long to run. Another natural extension is to vary the operating environment used for the study, including the Java VM parameters, the bus architecture used for secondary storage, and the number of processors used.

The most important area of further investigation is to attempt to separately measure worst-case, average-case, and best-case access patterns for object creation, updating, and retrieval. This requires developing algorithms that simulate a variety of access patterns and measuring the performance impact of each one.

Another important area of study required is the effect of access methods and caching on performance. The performance impact of the web services layer, which is likely to be significant, needs to be studied in order to develop effective techniques to mitigate it (these include various forms of batch processing). Caching strategies need to be investigated to determine which strategies are optimal for various operating environments.

Finally, the use of database backends other than MySQL needs to be investigated. The NMDS implementation currently supports Oracle and Sybase in addition to MySQL.

7 Conclusion

This study provides a preliminary performance analysis of the NEES metadata service implementation. The results of this study are the beginning of the establishment of a set of baseline performance measurements against which future refinements of the NMDS implementation, or deployments of the NMDS in different operating environments, can be assessed.