A Scalable Architecture for Maximizing Concurrency

Jonathan M. Crawford
Lockheed Martin Space Operations
Upper Marlboro, MD 20774
jcrawfor@eos.hitc.com
tel +1-301-925-1074
fax +1-301-925-0651

Abstract

This paper describes a design that addresses limitations inherent in the initial implementation of the Archive for the Earth Observing Systems (EOS). The design consists of two elements: a Request Manager client interface and a Thread Manager design pattern. In combination, these design elements provide dramatic improvements in reliability and scalability. Moving to a transaction processing-based design also provides substantial gains for operability.

1 Introduction

The EOS Archive is a multi-site distributed data warehouse of Earth-oriented satellite images and science algorithms/reports. Data holdings in the archive are expected to exceed two petabytes by 2002 [1]. Within the EOS Core System (ECS), the Storage Management (STMGT) Configuration Item has the sole responsibility for moving data into and out of the archive, and for both the physical and electronic distribution of media.

The projected load on the system consists of over 76,000 granules per day – more than two terabytes – inbound to the archive, and over 30,000 granules per day of retrieved data. Each granule consists of one or more image files plus a metadata file. In aggregate, more than 400,000 requests will be serviced daily by STMGT servers. These requests are prioritized, and must be serviced in priority order, particularly as system resources become increasingly scarce.

The ECS software uses Distributed Computing Environment (DCE) Remote Procedure Calls (RPCs) as the mechanism for inter-process communications, with software processes deployed on both Sun and SGI platforms. Server processes are multi-threaded (using DCE threads), and accept requests from client processes via synchronous RPCs (see Fig. 1). Since most STMGT operations are I/O-intensive, these tend to be long-

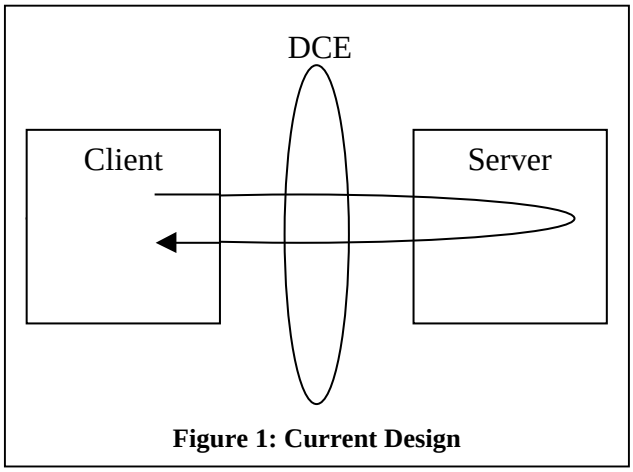

Figure 1: Current Design

running RPCs. Deployment of ECS has revealed significant scalability issues with the current STMGT implementation arising out of the use of synchronous DCE connections. DCE sizes the inbound request queue based on the number of listen threads available. Servers must either be configured with a large number of listen threads – consuming proportionately increasing system resources – or the number of concurrent requests must be limited, risking queue overflow and jeopardizing the system's ability to service the targeted request volume levels [2]. In some cases, the optimal number of listen threads is proportionate to the available resources. For example, a server that generates media for physical distribution of acquired data ideally has the number of listen threads equal to the number of devices available for generating media. However, due to the queue depth restrictions imposed by DCE, the number of listen threads must be configured to reflect the request queue depth, which may be considerably greater than the available resources.

2 Solution Elements

Our challenge was to redesign STMGT to improve scalability without disrupting client interfaces. At the same time, we sought to improve reliability and operability. This effort led to two major design elements: Request Manager, a means of preserving the existing client interfaces; and Thread Manager, a design pattern with highly desirable properties.

2.1 Request Manager

Request Manager provides the only DCE-based interface into STMGT. Existing STMGT client libraries were re-implemented to format requests as database transactions and submit them to the Request Manager via a single RPC. Request Manager executes each transaction, which checkpoints the request to a relational database and returns the server chosen to service the request. A single, centralized database is used for all STMGT requests, though multiple Request Manager processes may be used for request checkpointing and routing.

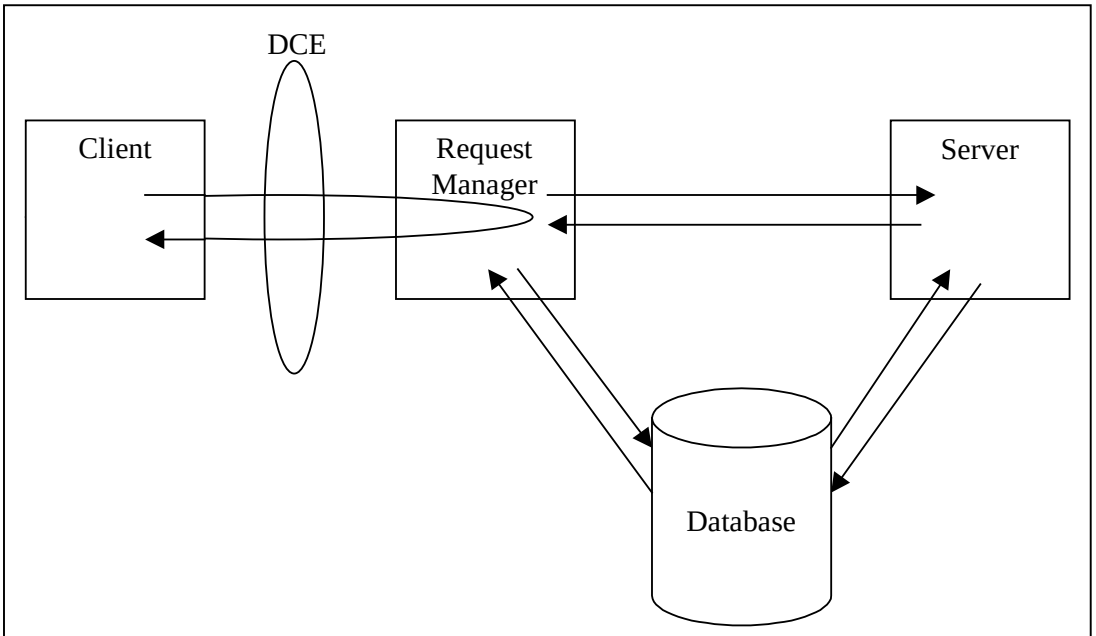

Figure 2: Request Manager Design

By using the database as the request queue, the Request Manager permits virtually unlimited requests to be queued. Moreover, priority-based dequeueing can be enforced

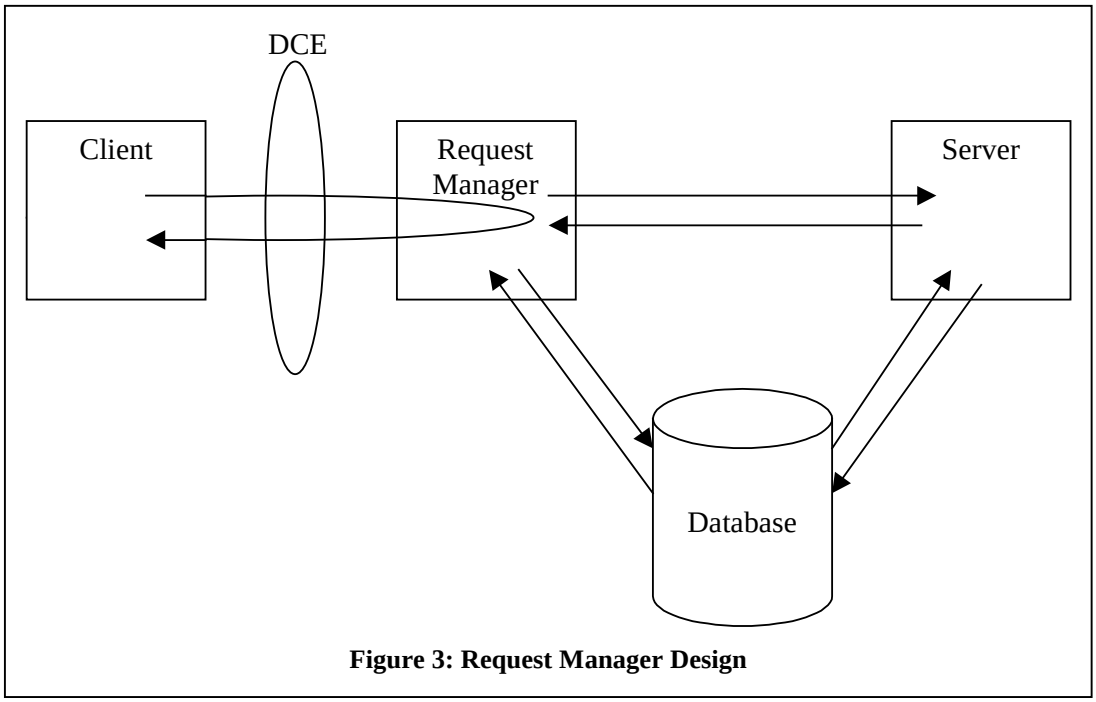

Figure 3: Request Manager Design

via the SQL stored procedures that are used to select the next available request for servicing. Stored procedures are also used to implement the routing logic, permitting adjustments to routing to be made strictly through database patches.

Once the server or servers have been identified for request processing, the Request Manager notifies each asynchronously using standard TCP/IP socket connections. No data is actually passed across the socket connection; the socket connection is merely used as an asynchronous event to notify the remote server that work is ready for processing. Figure 2 illustrates the interfaces within the new design.

2.2 The Thread Manager Design Pattern

The Thread Manager design pattern is comprised of four elements: a receptionist, a manager thread, and a pool of service threads. Upon creation, the receptionist allocates a port for listening and registers the server in the database as available for work. When the Request Manager or other STMGT servers assign work to the server, they notify the server by establishing a socket connection on the port on which the receptionist is listening. The receptionist then wakes the manager thread via a process-shared condition variable.

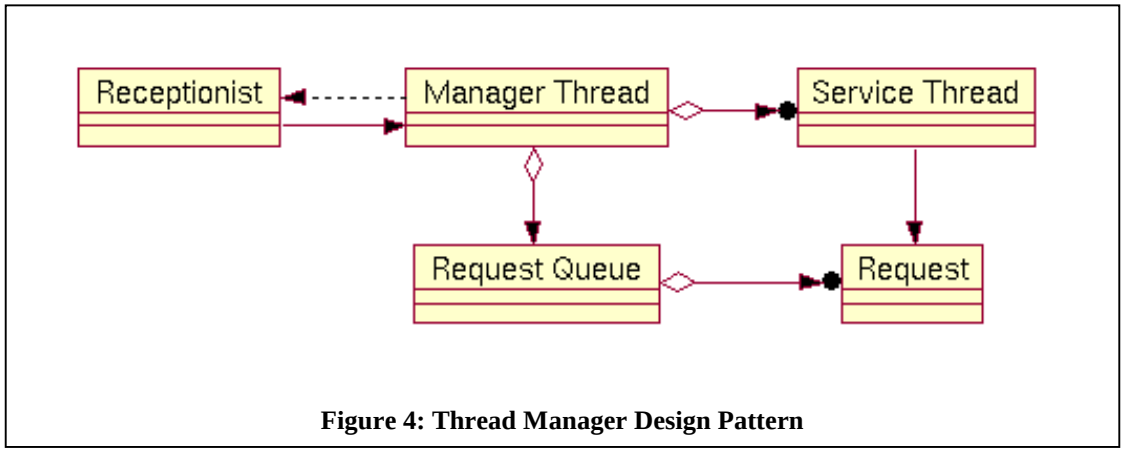

Figure 4: Thread Manager Design Pattern

The manager thread controls the assignment of requests from the queue to the pool of service threads. It waits on a condition variable until such time as it is notified by the receptionist that new work has arrived in the queue, or by a service thread that the service thread has completed its work and is ready for a new assignment. Restrictions are placed neither on the number of requests in the queue, nor on the number of service threads available for assignment.

Requests are assigned to service threads, which identify the type of work specified by the request and do the necessary processing to service the request. Service threads checkpoint as they perform request processing. If a request reaches a state whereby further processing must wait for another task to complete, the service thread will checkpoint the request in its current state and move on to another request. Once the related task has completed, the original request is released and re-assigned by the manager thread to a service thread.

In the Thread Manager design pattern, the pool of service threads is allocated at startup, and each service thread remains alive, even if it is idle. This eliminates the overhead of creating and destroying threads each time a request arrives. The number of service threads allocated to the pool need not be related to the expected volume of requests. For

example, in media distribution servers, the number of service threads typically equals the number of devices available for writing. Each service thread may also be assigned a minimum priority for servicing, thus ensuring that a glut of lower priority requests does not starve higher priority requests.

3 Benefits

Software failover for STMGT is now supported through simple database mechanisms. When a server is brought down, a monitoring process automatically identifies the server as unavailable. When the alternate server instance is brought up, its receptionist registers it as available for work. Since the Request Manager relies on stored procedures to perform request routing, subsequent requests are automatically routed to the alternate server instance. On server termination, the manager thread can also be implemented to un-assign any requests associated with that server, thus enabling the dynamic reassignment of in-progress requests to alternate server instances.

The Request Manager design redefined request submission in terms of a checkpoint to the database, and the Thread Manager design pattern enforces checkpointing at each request state. This ensures maximal recoverability in the event of failure, while minimizing reprocessing.

In the prior implementation, clients notified each server when they were restarted after termination, in order to reclaim any persistent resources that may have been leaked. This caused particular problems for clients which retrieve data from the distributed archive, since such processes talk to a variety of server instances, and often had no record of which server instances they were in communication with at the time of termination [3]. With the Request Manager/Thread Manager approach, the client notifies the Request Manager, which then notifies all affected servers based on any requests that are still in the database for that client.

The Thread Manager design pattern provides a request dependency table. This permits requests to be related, so that processing of one request is suspended until one or more other requests have completed processing. When a request becomes dependent on another request, the service thread can abandon processing of the dependent request and work off other requests. This dependency tracking also eliminates redundant I/O, such as when multiple requests arrive concurrently to copy or FTP the same file.

The Thread Manager approach uses no polling, reducing the CPU load. It also relies on asynchronous communications between STMGT servers, eliminating the blocking and associated resource usage associated with long-running RPCs. Service threads are never left blocked, occupying memory resources or swap space and preventing other requests from executing while the blocked thread is effectively idle. In essence, the design is such that requests seek the I/O bottlenecks. Processing rates are anticipated to approximate the maximum hardware throughput supported by the physical configuration.

The centralized request tracking provides unprecedented operability gains. The STMGT GUI can now observe the processing progress of every request in every server. Future

enhancements can utilize expected processing times to rapidly and automatically detect requests which are "frozen."

4 Future Work

The Request Manager was introduced as a process solely in order to preserve the existing, synchronous, DCE-based client interfaces to STMGT servers. However, the logic to checkpoint requests to the database and notify appropriate servers is encapsulated in base classes which are integrated into the Thread Manager design. All STMGT servers now use asynchronous client interfaces to communicate within STMGT, bypassing the Request Manager process entirely. As external clients to STMGT servers are able to adapt to an asynchronous request processing model, the Request Manager may eventually be removed altogether, supplanted by the set of asynchronous client interfaces which checkpoint requests directly to the database.

5 Conclusions

By using a transaction processing model within STMGT, we have dramatically increased both the operator's and our ability to monitor system activity. We expect to leverage these operational benefits by capturing check-pointed event sequences for regression testing and debugging. This ability to reproduce site-specific error conditions is expected to enable us to provide unprecedented turnaround time to correct complex errors that manifest at the deployed sites.

By replacing the DCE-based architecture with the Thread Manager design pattern, we were able to dramatically improve the scalability and reliability of ECS with regard to its storage management capabilities. As ECS continues to deploy satellites which feed the EOS Archive, the stability and reliability of STMGT will become increasingly crucial; we believe this new design will ensure the continued success of the ECS system.

5 Acknowledgements

The author would like to thank Donald Brown of Lockheed Martin Space Operations for his support and technical insights in the development of this new design.

References

[1] *Functional and Performance Requirements Specification for the Earth Observing System Data and Information System (EOSDIS) Core System* (http://spsosun.gsfc.nasa.gov/ESDIS_Pub.html, 1999) Appendix C.

[2] *OSF DCE Application Development Guide – Core Components, Rev. 1.1* (Cambridge, MA: Open Software Foundation, 1994).

[3] A Lake, J Crawford, R Simanowith and B Koenig. "Fault Tolerant Design in the Earth Orbiting Systems Archive", *Eighth NASA Goddard Conference on Mass Storage Systems and Technologies* (2000).