Experiences With an Object Manager for a Process–Centered Environment

Dennis Heimbigner
Computer Science Dept.
University of Colorado
Boulder, CO 80309-0430

Abstract

Process-centered software engineering environments, such as Arcadia, impose a variety of requirements on database technology that to date have not been well supported by available object-oriented databases. Some of these requirements include multi-language access and sharing, support for independent relations, and support for triggers. Triton is an object-oriented database management system designed to support the Arcadia software engineering environment. It can be used as a general purpose DBMS, although it has specialized features to support the software process capabilities in Arcadia in the form of the APPL/A [Sut90] language. Triton was developed as prototype to explore the requirements for software environments and to provide prototypical solutions. By making these requirements known it is hoped that better solutions will eventually be provided by the database community.

This material is based upon work sponsored by the Defense Advanced Research Projects Agency under Grant Number MDA972-91-J-1012.

**Proceedings of the 18th VLDB Conference
Vancouver, British Columbia, Canada 1992**

1 Introduction

By now, there is a general understanding that object management[1] is central to software engineering environments. It serves as one of the primary means for integrating components of the environment by providing a common set of data structures (schema) and a shared repository for persistent objects. A sufficiently dynamic object manager is also important in providing extensibility in an environment by allowing incremental extensions to the schema and hence to the range of tools that can share information.

The Arcadia project [Kad92, TBC+88] is constructing an environment that is one of the first of a new class of so-called software-process centered (or software-process driven) engineering environments. A process centered environment is one in which the programmer is guided in the task of producing software according to some methodology. Such an environment extends the more traditional tool-oriented environment by adding the capability to specify the process by which software is to be constructed. This is in contrast to a typical tool based environment in which the programmer is presented only with a collection of tools and is given no help in deciding how to apply those tools to produce a software product.

It is assumed that a process-centered environment will be controlled by a model of the process written in some formalism. Osterweil [Ost87] has proposed the use of an executable programming language as that formalism. Such a language is called a process programming language (PPL). Arcadia uses a process programming language approach as the basis for its environment.

Arcadia has been active in object management since its inception. As its environment has evolved, it has exposed a number of requirements for object man-

[1] The software engineering community tends to use terms such as object management rather than terms such as database management. But the various terms should be considered essentially interchangeable.

agement needed to support a process centered environment. The Arcadia approach has consistently been to use existing database systems and to augment them with innovative features as necessary for meeting those requirements. We (Arcadia) have taken this approach because we have found that database research prototypes[2] provide many, but not all, of the capabilities needed to support a process-centered environment. Modern database systems are very large and complicated pieces of software, spending tremendous amounts of effort to furnish reliable, secure, efficient storage management and concurrency control. Different research database prototypes provide, in addition, such features as schema dynamism, long transactions, and very large object management. Arcadia has no wish to duplicate these capabilities. Unfortunately, at the time that Arcadia needed them, no single database system provided a combination of these capabilities sufficient to support a process-centered environment. Even today, it is not clear if any such system exists. Our approach has been to augment existing systems in order both to provide increasingly satisfactory object management in Arcadia and to gain clearer understanding of the requirements for object management in a process-centered environment.

Triton is one of the object managers in use in Arcadia. It has been our primary vehicle for exploring the needs of the process-related activities within the Arcadia environment[3]. Triton may be briefly characterized as a general purpose object-oriented database system. More specifically, it is a serverized repository providing persistent storage for typed objects, plus functions for manipulating those objects.

It is important to understand that Triton is not being held out as a serious competitor to other databases being developed in the database community; it has a number of flaws in performance, robustness, and data model. Rather, it is a vehicle for demonstrating possible solutions for the needs of software environments. The primary thrust of this report is to describe those requirements and the solutions embodied in Triton with the hope that the database community will incorporate better solutions in their next round of research prototypes.

Triton uses an existing system, Exodus [CDG+90], to provide much of its functionality (basic type model, buffering, persistence, etc.). Exodus was originally characterized as a database toolkit [CDF+86] where a database implementor used the elements of Exodus to build a custom DBMS. In practice, Exodus is better characterized as a persistent programming language system. It consists of a storage manager and a persistent programming language named E [RC87]. E may be considered as a persistent version of C++, and like C++ it has the C type system augmented by classes with behaviorally defined methods. The original expectation, as with most persistent programming language systems, was that application programs would be implemented as E programs and the collection of such programs would constitute a customized DBMS. Triton adds value to Exodus by adding features needed in Arcadia, but not directly provided by Exodus. Some of these additions, such as triggers, are driven by the needs of process support and some by the problem of ger ralizing persistent programming language systems to support software environments.

This report may occasionally seem to be overly critical of Exodus, but this is misleading. In many ways, the Triton effort pushes Exodus in directions for which it was never designed. We understand that its goals were not our goals and so mismatches should not be surprising. Exodus has performed well in Triton and it has demonstrated a remarkable degree of flexibility in meeting the requirements we imposed upon it.

In this paper, we will address the requirements for software environments in the context of describing the architecture and associated rationale for Triton. We will then show some of its features: heterogeneity, the interface, dynamic definition, process language support, and triggers. Finally, we will describe our observations, insights, and lessons gained in the process of constructing and using Triton.

2 Requirements

The initial design of Triton was influenced by three general requirements that we felt were essential to support any process-centered environment.

- Efficient representation for the wide variety of software artifacts used within Arcadia: abstract syntax graphs, requirements and design nodes, configuration management graphs, test cases, documentation, and so on. In practice, the software community believes that only some form of behavioral object-oriented model extended with relations is sufficient.
- Support for process coding languages-especially APPL/A. This requires a system supporting at least relations and triggers, or some equivalent form of event notification.
- Standard and non-standard database concurrency and recovery mechanisms. This issue will not be

[2] The issue of commercial systems is addressed in section 9.

[3] Arcadia uses several object managers for a variety of purposes. See [WWFT88] for information about another object management activity within the Arcadia project.

addressed in this paper.

As the Triton project progressed, our understanding of the problems of object management deepened. The general requirements were elaborated and additional requirements were added to reflect our increased knowledge.

Multi-language interoperability is such an additional requirement. This requirement has become increasingly important as Arcadia has evolved and we now view this requirement as one that is co-equal in importance with process support. This requirement stems from a variety of constraints. For example, many DOD programs mandate the use of Ada as the language for implementing systems. Many artificial intelligence systems, such as AP5 [Coh88], already have a heavy investment in Lisp, but could benefit from access to a object base.

Additionally, we have seen that a single persistent programming language interface makes too many assumptions that turn out to be invalid for one or another application. For example, Arcadia uses two rather different models of persistence: persistence by type (as in E), and persistence by instance (as in PGraphite [WWFT88]. Both of these models have good justifications in terms of the applications that use them, but it is rare to find a database system that supports more than one model of persistence.

Multi-language interoperability covers two capabilities. First, we require the object manager to be accessible from programs written in a variety of programming languages. Currently, Arcadia has components written using Ada, C, C++, Lisp, and Prolog.

Second, it must be possible for programs written in various languages to share data. There are two typical ways to achieve sharing: (1) pair-wise conversion or (2) use of a common data model. We rejected choice one as being ultimately too time consuming and chose instead to use the common data model approach, even at the expense of such problems as incomplete model mappings. 28 The multi-language issue is part of the larger issue of support for general heterogeneity. Arcadia has had to face not only language heterogeneity but heterogeneity of machine architectures and compiler heterogeneity (same language, same machine, but different compilers). Many of the issues that we first identified with multiple languages in fact can appear even within a single language. To state it strongly, interoperability over heterogeneous systems has now become a driving factor in the architecture of Triton.

Dynamic definition of schema elements (including object methods) is another requirement for Triton. It is assumed that although any one application may use a stable schema, new applications will be continually added to the environment, thus requiring new schemas. A catalog (or meta-database, or data dictionary) is a necessary corollary of dynamic schema definition. The catalog is needed to define the known schema and to allow browsing of existing schemas. In many database systems, this requirement would seem to be automatically provided. Unfortunately, Exodus, like many persistent programming language systems, did not have (or need) this capability, and so it became an additional problem for us to address.

In addition to purely technical requirements, there was a requirement to reuse as much existing software as possible. If constructed from scratch, Triton would have taken too many resources to be practical. So from the outset, it was important to avoid re-implementation. As a consequence, Triton was constructed using as much existing database technology as possible. It was important to focus the Triton effort onto those features essential to Arcadia and to reuse those components that were properly the domain of the database community.

Obviously, a number of desirable capabilities (e.g., versioning), are missing from this list of requirements. But, at the time that the Triton project started (in early 1989), no system that was obtainable appeared capable of completely satisfying even this minimal set of requirements. We could find systems that had, for example, transaction management and triggers, but that were only accessible through a fixed programming language, or had difficulty with dynamic type creation. We decided that our only recourse was obtain a database manager offering a close match to our needs and to modify it.

3 Overview of the Triton Architecture

Figure 1 shows the architecture of Triton. It is a client-server architecture in which the client communicates with the server using a Remote Procedure Call (RPC) protocol. In this case, we use Q [MS89], which is a variant of the Sun RPC/XDR protocol that has some adaptations for multi-language interoperability. This is indicated in the figure by the arrow labeled "Q". It represents a "calls" relationship between client and server. Discussion of the architecture of the client shown in the figure will be deferred to section 7. Suffice it to say that it communicates using RPC to call the interface functions provided by the server.

The server has five major components.

1. The *server interface* handles the details of receiving requests from clients (there may be more than one), invoking the appropriate local procedure to

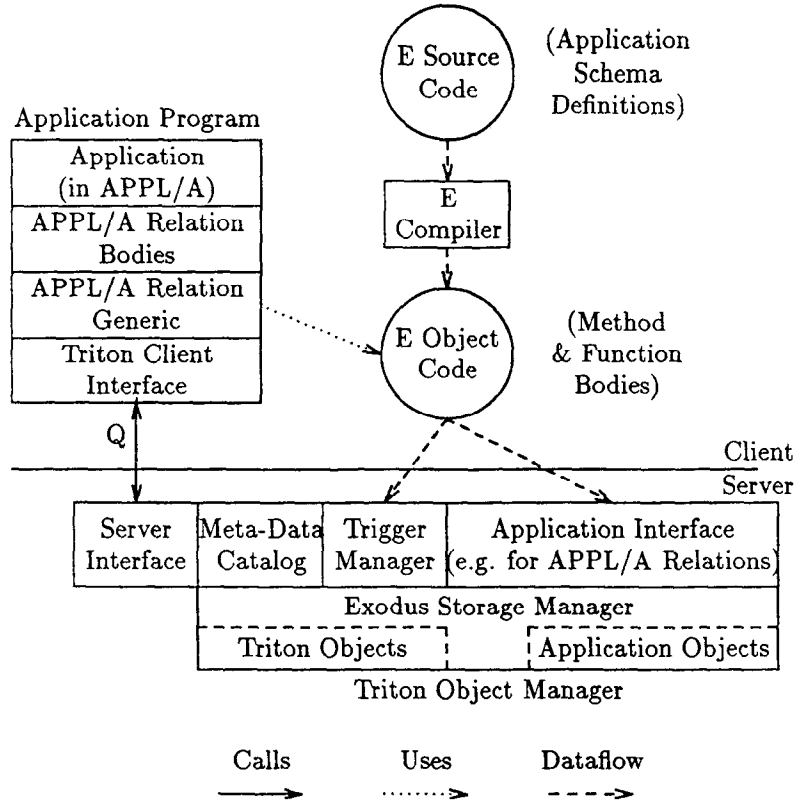

Figure 1: Triton Architecture.

field the request, and returning any result back to the client.

2. The *catalog* component is a meta-database written as a collection of E types. It records the structure of the schema currently known to the server.

3. The *trigger manager* interacts closely with the catalog and manages the attachment of triggers to various schema elements and their subsequent invocation.

4. The *application interface* is the collection of methods and functions defined by client applications.

5. The *Exodus Storage Manager* provides for the persistent storage of objects from the other components. In particular, it will store any data objects defined by Triton itself: trigger references and catalog objects, for example. Additionally, the storage manager stores actual application objects.

4 The Triton Interface

The Triton server presents a procedural interface to its clients. That is, to a client it "looks" like a library of procedures for manipulating schema elements and objects. Triton makes significant use of *handles*, which are references to objects in the server. The client can only get handles from server, copy them around, and send them as arguments back to the server. The client has no knowledge of the internal structure (if any) of the handles.

Manipulating and accessing the Triton catalog represents a significant portion of the operations provided by the server. The Triton Catalog provides two major capabilities.

Schema Definition: These operations allow a client to dynamically define schema elements into the catalog. Many operations return a handle to the defined schema element. The definable schema elements are classes, methods, functions, and formal arguments to methods and functions. There are corresponding schema operations to *destroy* elements, but their semantics are admittedly not well-defined.

Name Space: The space of schema elements in the catalog is almost flat. At the top level are uniquely named classes and functions. Classes "contain" named methods, and methods and functions "contain" named formal arguments. The name space operations allow clients to con-

vert a name of a schema element into a handle for that element.

The primary actions of the interface code are to receive requests from clients to invoke methods or functions defined in the catalog. The typical E operation of invoking a method (roughly speaking, *results = Instance.method(inputs)*) is mirrored by specific operations in the interface.

5 Managing Client-Server Heterogeneity

Triton is designed to operate in a heterogeneous environment. Heterogeneous in this case refers to differing machine architectures and/or differing client side programming languages. Minimally, it is assumed that it is possible to have TCP/IP connections between the client and the Triton server. In order to understand some of the Triton interface, it is necessary to understand how Triton manages these various kinds of heterogeneity.

Managing multi-language access and sharing has had a pervasive influence on Triton. Multi-language data sharing is achieved using a common data model. A subset of E, the Exodus persistent programming language, is used as the common data model. We examined and rejected the idea of defining a new model as too resource intensive. The supported E-subset model includes basic types such as *integer* and *float*, as well as constructors such as *struct* and *class*.

The major problem in using a common data model is data model mapping between the languages and types defined in the client and the types defined in the server. For any given type in the client language there must be a mapping to some equivalent E type(s). In practice, we have had little difficulty finding a reasonably large subset of E types that can be mapped to and from the other language type systems (see [Hei90]).

Multi-language access to Triton is provided by a standard remote procedure call protocol (Q) on top of TCP/IP. Q is designed to make access between Ada and C as simple as possible, and defines standard mappings between a subset of C types and a subset of Ada types. A remote procedure call operates by *marshaling* the inputs to the procedure on the client side. That data is sent to the server along with some handle specifying the remote procedure to be invoked. The server *unmarshals* that data, and calls the appropriate procedure. It then takes any result, marshals it and returns it to the client. Without going into details, suffice it to say that any marshaling mechanism must provide two features: (1) linearization of arbitrary data structures and (2) a standard intermediate data representation.

The requirement for multi-language access was critical in determining the final Triton architecture. Experience shows that we could not count on being able to place any portion of the code for the object manager (including method code) in the client (see section 8.2 for more on this issue). As a result, we settled on the client-server architecture of Figure 1 with a Triton server residing in one Unix process and each client residing in a separate Unix process. This architecture closely resembles the original Gemstone [BOS91] architecture and the proposed Thor [Lis92] architecture. This separation solves many of the multi-language problems by placing the Triton system in one address space and restricting other language programs to separate address spaces. In principle, any language that can support RPC can use the Triton server.

6 Dynamic Definition

In E, a method (or function) schema element is defined behaviorally. That is, it is associated with a piece of code that is executed when the method is invoked via an interface operation of the server. Methods may be dynamically defined in the catalog and so Triton must have some means for dynamically obtaining the code associated with the method.

Our approach is to dynamically load compiled E code into the Triton server. This is represented in Figure 1 by the column of objects in the upper right. It shows E source code for the methods as input to the E compiler, which produces E object code. This code is loaded into the server to define methods, functions, and triggers.

For this to work, Triton requires the use of a dynamic loader. For the details behind dynamic loading, see [Hei90]. After the method has been loaded, subsequent invocations are direct. Additionally, it is possible (and common) to load multiple methods at one time, thus speeding up the process considerably.

Unfortunately, in Triton there are two definitions of the structure of, for example, a class type. One is the structure defined in the catalog. The other is the structure implicit in the compiled method code. It is possible to have inconsistencies between the two definitions, and this is, of course, deprecated. When and if the server takes more control over the source code for methods, this inconsistency will be eliminated.

7 Triton Support for Process

The key feature of a process centered environment is its ability to support and enforce processes for

constructing software. It is assumed that multiple processes may be defined using a so-called "process programming" (or "process coding") language. APPL/A [Sut90, SHO90] is a prototype of one such process programming language. It is defined as an extension to Ada [Uni83].

There is some consensus in the software engineering community that object management support for process (and the products produced by the process) entails at least the following features: relations, triggers, constraints, and non-standard transactions. All of these are present in some form in APPL/A, and the first two have direct representations in Triton. The last two are subjects of ongoing research in Arcadia and will not be discussed further[4]. Section 8 will provide some rationale for relations and triggers. As an aside, we note that each of these features can be used for other purposes; none of these capabilities is only for supporting process.

This section will briefly describe the Triton support for APPL/A relations and triggers. We do assume some familiarity with Ada. Note that the relation support is built on top of the object-oriented data model of Triton; it represents a notational extension to the model.

7.1 APPL/A Relation Support

In APPL/A, a relation looks like a combination of an Ada *package* and an Ada *task*. It defines the structure of the relation tuple and a limited set of operations: insert, delete, update, and find. This last operation (find) is used to provide a combination of tuple-at-at-time access and associative retrieval. As with Ada packages, a relation definitions has two pieces: a specification and a body. The body is expected to provide implementations for the interface operations defined in the specification. See [Sut90] for details.

Referring back to Figure 1, we can now examine the client architecture. The client shown there reflects the various layers required by an APPL/A program to communicate with the server. The top level application is defined in terms of a collection of APPL/A relation specifications. These specifications are implemented by relation body code. These bodies use the APPL/A generic relation interface, which is in turn implemented by a generic body. This generic body is defined using the Triton client interface library.

On the server side, an APPL/A relation is defined by a corresponding E class providing methods matching the APPL/A operations of find, insert, update, delete, and whose is a set of tuples. The dotted arrow in Figure 1 from the APPL/A generic interface to the E-code represents this correspondence. The Triton reference manual [Hei90] may be consulted for the details.

7.2 APPL/A Trigger Support

Triton has augmented the E capabilities with a simple form of trigger. As might be expected, a trigger is a piece of code that is invoked whenever some *event* occurs[5]. In Triton, the events that can invoke triggers are (1) method or function invocation and (2) method or function completion. There are important restrictions on trigger attachment.

- Only methods or functions invoked via the evaluate_method or evaluate_function interface calls can cause triggering. Thus, an internal call from one function to another will not cause triggering.

- Instance specific triggers are not provided. If a trigger is associated with a method, then every invocation of that method for all instances will invoke the attached trigger.

8 Some Lessons from the Triton Experience

Constructing Triton has been an enlightening experience. It has done much to clarify the actual requirements for object managers when they are expected to support a process-centered environment. The following sections elaborates on some of the lessons that we learned from the Triton effort and provides additional rationale for the architectural choices outlined in previous sections.

8.1 Using A Persistent Programming Language

We seriously underestimated the amount of effort that it would take to use a persistent programming language system as the basis for Triton. In retrospect, this may not be surprising since such systems were never designed to operate in a client-server environment. But in defense of this activity, it is important to note that in recent years in the database community (and in the commercial word as well),

[4]The reason is that the APPL/A transaction model is more general than can be supported by any currently available system, research or commercial.

[5]A state-based approach, as in AP5 [Coh88], is also possible in which the trigger is invoked when some defined *system state* is reached. Triton does not support this style of trigger.

many new research efforts have assumed a persistent programming language as their basic architecture [RC87, TW91, D+91, LLOW91]. The assumption implicit in this approach is that all the programs that access stored data will be written in whatever persistent programming language has been chosen. Even in the cases [D+91, TW91] where multiple languages are, in theory, supported, there is no obvious provision for sharing data between those language.

In any case, this language-specific approach turns out to be completely incorrect for an environment such as Arcadia. Multiple languages sharing data is the norm, not the exception. As we have seen with Triton, converting a language specific system to a more general system has a number of painful consequences:

1. One is almost compelled to use the persistent programming language as the basis for the common data model for the system. But, the problem is that the type system of this language may be much more complicated than is necessary. Additionally, some of the features of the language (such as generics) have complicated implementations that are difficult to model in the catalog.

2. Overgeneralizing somewhat, it is typical for language specific systems to assume a relatively static type schema. It is often assumed that all the schema information is compiled into programs. Converting to a dynamically defined schema appears to require some form of dynamic loading of compiled code. This in turn requires significant support from the compilers, loaders, and even possibly the operating system. Porting Triton out of a narrow range of Berkeley Unix class of systems, for example, would be a daunting task.

8.2 Separate Client and Server Address Spaces

In our experience, it is a serious mistake to assume that one can load any component of the object management system in the same address space as client code. Especially if the client code is written in a language different from the language used for the object manager. Run-time systems often make assumptions about their control over such things as signals, memory allocation, and file descriptors. Our sad conclusion is that mixing run-time language support systems in the same address space will fail more often than not[6]. Perhaps in some distant future, there will be standards for run-time systems, but until then, code mixing is fraught with peril.

This dictum also applies to loading behavioral methods into the client address space. It has been proposed [Mai91] that the server should keep either interpretive versions of method bodies, or per-language versions of compiled method bodies that can be loaded into the client as needed. The use of interpretive models should work, although it might require the construction of an interpreter written in each supported language, which may bring back all of the issues of run-time contention. We are sceptical about the use of any form of compiled code in a client. This requires the inclusion of dynamic loading mechanisms into each client, and we believe that run-time contention will cause this to fail.

The alternative used in Triton keeps the object manager code plus the method body code in a separate address space. We recognize some of the costs involved in this approach; there are significant performance hits in transporting method inputs to the server and retrieving the outputs. This may in some cases be offset by the more efficient execution of the methods since they are closer to the data.

8.3 Common Data Model

Triton achieves multi-language interoperability by using a common data model. Client data is converted to the common model and stored in the server. On retrieval, the data in common format is converted back to the client model. In spite of our problems in using the rather ugly E/C++ type model, we were pleasantly surprised at how well this approach worked in practice. We hypothesize that the client-server architecture was the "cause." That is, inherently when a client sends data to the server, it must convert the data to a standard linear form in order to ship it over a thin-wire connection to the server. Adding a little additional complexity to convert to and from the common data model is not a large burden. Given a shared memory model of client-server communication, the cost of conversion might seem more onerous.

8.4 Support for Relations

The software products managed by processes often take the form of a collection of objects with a variety of graphs superimposed over those objects. It is important to realize that the set of all graphs may not be known a-priori[7]. For example, the nodes of an abstract syntax tree may need to be *annotated*

[6] In fact, contention can appear even with our minimal RPC support in the client. This happened with Ada and caused serious problems.

[7] There is an obvious correspondence to the Schema evolution problem.

with additional information for purposes of analysis. As new analysis techniques are defined, new annotations may be needed. Relations, defined over node-types, but independent of (i.e., not known to) the node type have proven to be extremely useful in representing such annotations, a fact that others have recognized [Rum87][8].

Our concern for relations has often been dismissed out-of-hand by proponents of pure object-oriented languages. The claim is usually made that the user can just define relations using the object-oriented type system, and technically that view is correct. The issue is more one of notation than power. Languages such as APPL/A rely heavily on relations as a major structuring element in their type system. Languages such as E make defining and using relations inconvenient at best because they require the programmer to build up even elementary support elements such as selection, indexing, and cursors that are needed to support a relational interface. In order to aid APPL/A based process support, the Triton interface was substantially augmented to provide such support elements for the definition and manipulation of relations.

8.5 Support for Triggers and Event Management

Experiences in defining processes makes it clear that some equivalent of triggers is required as a means for responding to unanticipated events (e.g., an emergency change to a software requirement) and as a means for unobtrusive monitoring of process activities (e.g., to measure the productivity of a programmer).

The trigger system in Triton may be considered a failure. As described in section 7.2, the trigger capabilities in Triton had too many limitations to meet the original requirement to support APPL/A triggers. It required trigger code to be rewritten from APPL/A to E and, more importantly, there was no mechanism for triggers to communicate back to clients asynchronously.

As the Arcadia environment has evolved, a more important problem has surfaced: it is not clear that triggers even represent the correct abstraction. Most environments are moving to use a more general notion of "event" as a replacement for triggers. Control integration via events (as in Field [Rei90] and HP-SoftBench [Hew89]) is rapidly becoming the norm.

It is important to note that event systems are distinct from the database notions of triggers and rules. In an event system, there is an event server (or dispatcher) to which programs send messages representing postings of events. Other programs may register with the dispatcher to receive events that match some specified pattern. As events arrive the dispatcher forwards them to registered programs as appropriate. The key feature here is that the dispatcher has very little knowledge about the senders and receivers of events or about the semantics of the events themselves. This results in a very flexible system in which new kinds of events may be posted dynamically and senders and receivers of events may come and go quickly. It is also important to note that the event dispatcher system is independent of any database in the system.

Trigger systems, as represented in most databases, often assume that only a fixed set of actions (e.g., object insertions and modifications) can generate events. More importantly, it is assumed that the action to be taken on event occurrence is known to the database system. With the possible exception of HIPAC [MD89], database systems appear ill-prepared to export their events to an external event dispatcher or to receive externally generated events.

8.6 Performance

The performance of Triton leaves much to be desired. And it is almost certainly true that better application of techniques already known in the database community could significantly increase the performance of Triton. If the strict client-server split is maintained, then there are some limits on the performance. On a Sun3, using Q and UDP, performing an evaluate_function on a function with an empty body and with no input or output takes about 20 milliseconds for a round trip. The primary costs are for RPC overhead and for catalog reference overhead. Use of shared memory, faster networks, and better protocols could help reduce the RPC cost. Reducing the catalog reference cost is also possible.

9 Alternatives

At the time that Triton was first conceived, there were only a limited set of acceptable choices on which to base the effort. Right from the beginning, commercial systems were excluded from consideration both for licensing reasons as well as lack of access to source code. This left only research vehicles to consider. This set was very small: apparently only Exodus and Postgres. We were intrigued by Postgres, but we had concerns that simulating an object-oriented database over relations (even the extended relations of Postgres) could have some performance problems. In fact, the issue

[8] One could use other structures, such as functions, as long as independent definition was maintained. Functions, however, would not be as useful in bi-directional query.

was moot because Postgres did not become available until well after the Triton project was underway. This left the Exodus system [CDG+90] from Wisconsin as our choice. In retrospect, this turned out to be a good choice because Exodus was quite robust and support was reasonable (given the inevitable limits associated with any research effort).

At the current time, there are a number of systems that, with more or less work, could serve as replacements for Triton. O_2 [D+91] and the Texas Instruments OODB [TW91] are similar to Exodus in that they support one or more persistent programming languages. Presumably, with some work, these systems could be used in Triton in place of Exodus by using one of their languages as a common model and adapting the dynamic loader to work with that language.

Postgres [RS87, SR86, SK91] is now available, and if we had the resources, it would be interesting to rehost Triton onto Postgres. It obviously supports relations well, it has a catalog, and it has a form of trigger. The simulation cost question is still open. Additionally, we are unsure how well Postgres can deal with various kinds of heterogeneity.

We originally rejected commercial systems, and in revisiting the alternatives, the problems of license and source code still remain. But we can now see that most of the commercial object-oriented systems support the persistent programming language model, which makes them no better than Exodus for our purposes.

PCTE+ [GMT86] is a often considered by the software community to be the first choice as object manager for an environment. Simplifying somewhat, PCTE+ can be viewed as an augmented file system in which files can have contents and attributes, as well as links to other files. A link is a unidirectional pointer and two of them can combine to form a binary relation. PCTE+ also has a notion of trigger. Both Ada and C interfaces are provided for PCTE+, but interestingly, there does not seem to be any support for heterogeneous access to data across the two languages. Finally, PCTE+ is very coarse grained; it supports objects of the size of files and very small objects could suffer large penalties in space and speed of access.

GemStone [BOS91], although commercial, seem much more promising as an alternative to Triton. GemStone is derived from Smalltalk [GR83] and its interpretive nature would seem to make it possible to augment the system with the exact event mechanisms required. Interpretation also allows for dynamic loading. It is not clear if GemStone has support for full heterogeneous access. Also, getting the effect of generic relations might be a little difficult since Smalltalk does not appear to support that kind of polymorphism.

As a by-product of our examinations of various systems, we are beginning to define some "challenge problems" that can help us determine the utility of a system for our purposes. At the moment, we can articulate several such challenge problems.

1. For an arbitrary imperative language, show how one would write a schema browser/editor for the given database system.

2. For two arbitrary imperative language, show how one would write a program that creates a schema and data using one language, and then reads and prints (in some reasonable format) that same data using the other language.

3. Given two client programs, show how one can define a trigger after the two clients have begun, and then show how both clients can be made to activate that trigger.

4. Show what is involved in defining APPL/A relations to utilize the given database system.

5. Show how the given database system can export events to a Field-style broadcast message server, and define the set of events which can be generated. For extra credit, show how the database can effectively receive and use such external events.

The point of these problems is, of course, to see what is involved in multi-language access and sharing, trigger management, and relation support.

10 Status and Future of Triton

At the moment, a version of Triton with limited transaction features is running on Sun 3 and Sparc machines running Sun OS 4.1.1 or later. We are investigating the possibility of a Mach port.

Triton is used within Arcadia to support our current process programs, such as REBUS [SZH+91]. It has been exported to some external groups such as the STARS project.

Triton is undergoing a number of relatively short term enhancements:

- The next version of Exodus provides transaction management facilities and we are currently rehosting Triton to use these features.

- We are working to replace Triton triggers with more general event mechanisms.

- We are exploring alternative, and simpler, common data models to replace E.

- We are exploring performance enhancements.

11 Summary

Triton is one of the first attempts to provide comprehensive object management support for process-centered environments. It provides a behavioral object-oriented type model capable of supporting process programming languages. Triton also provides explicit support for heterogeneous interoperability in the form of multi-language access as well as shared data using a common type model. Implementing Triton has increased our understanding of the requirements for such object managers and we are now in a better position to inform the database community about those requirements. New object managers that address those requirements would then be appropriate candidates for inclusion in a process-centered environment.

Acknowledgments

I wish to thank the members of the Arcadia Consortium, and especially Lee Osterweil, for their insights. Mark Maybee, Hadar Ziv, Harry Yessayan, and Jose Duarte were especially helpful in testing out Triton and providing useful suggestions. I also wish to acknowledge the help of the Exodus project in providing a very useful system.

References

[BOS91] Paul Butterworth, Allen Otis, and Jacob Stein. The Gemstone Object Database Management System. *Communications of the ACM*, 34(10):64–77, October 1991.

[CDF+86] Michael J. Carey, David J. DeWitt, Daniel Frank, Goetz Graefe, Joel E. Richardson, Eugene J. Shekita, and M. Muralikrishna. The Architecture of the EXODUS Extensible DBMS: a Preliminary Report. Technical Report Computer Sciences Technical Report #644, University of Wisconsin, Madison, Computer Sciences Department, May 1986.

[CDG+90] Michael Carey, Dave Dewitt, Goetz Graefe, Doug Haight, Joel Richardson, David Schuh, E. Shekita, and S. Vandenberg. The EXODUS Extensible DBMS Project: an Overview. In Stan Zdonik and David Maier, editors, *Readings in Object-Oriented Databases*. Morgan Kaufmann, San Mateo, CA, 1990.

[Coh88] Don Cohen. *AP5 Manual*. Univ. of Southern California, Information Sciences Institute, March 1988.

[D+91] O. Deux et al. The O_2 System. *Communications of the ACM*, 34(10):34–48, October 1991.

[GMT86] Ferdinando Gallo, Regis Minot, and Ian Thomas. The Object Management System of PCTE as a Software Engineering Database Management System. In *Proc. Second ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 12 – 15, 1986.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[Hei90] Dennis Heimbigner. Triton Reference Manual. Technical Report CU-CS-483-90, University of Colorado, Department of Computer Science, Boulder, Colorado 80309, August 1990.

[Hew89] Hewlett-Packard. *HP Encapsulator: Integrating Applications into the HP SoftBench Platform*, 1989. HP Part No. B1626-90000.

[Kad92] R. Kadia. Programming Heterogeneous Transactions for SDE's. In *Proceedings of the Darpa Software Software Technology Conference*, pages 287–302, April 1992.

[Lis92] Barbara Liskov. Preliminary Design of the Thor Object-Oriented Database System. In *Proceedings of the Darpa Software Software Technology Conference*, pages 50–62, April 1992.

[LLOW91] Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. The Objectstore Database System. *Communications of the ACM*, 34(10):50–63, October 1991.

[Mai91] David Maier. Re: Looking for definition of OODB (an OODB manifesto). Message to comp.object news group, 28 August 1991.

[MD89] Dennis R. McCarthy and Umeshwar Dayal. The Architecture of An Active Data Base Management System. In *Proc. of the ACM SIGMOD International Conf.*

on the Management of Data, pages 215 – 224, 1989.

[MS89] Mark Maybee and Stephen D. Sykes. Q: Towards a multi-lingual interprocess communications model. Arcadia Document UCI-89-06, Department of Information and Computer Science, University of California, Irvine, Irvine, February 1989.

[Ost87] Leon J. Osterweil. Software Processes are Software Too. In *Proc. Ninth International Conference on Software Engineering*, 1987. Monterey, CA, March 30 – April 2, 1987.

[RC87] Joel E. Richardson and Michael J. Carey. Programming Constructs for Database System Implementation in EXODUS. In *Proc. ACM SIGMOD Conf.*, pages 208–219, 1987.

[Rei90] Steven P. Reiss. Connecting Tools Using Message Passing in the Field Environment. *IEEE Software*, pages 57–67, July 1990.

[RS87] Lawrence A. Rowe and Michael R. Stonebraker. The POSTGRES Data Model. In *Proc. of the Thirteenth International Conf. on Very Large Data Bases*, pages 83 – 96, 1987.

[Rum87] James Rumbaugh. Relations as Semantic Constructs in an Object–Oriented language. In *OOPSLA'87*, pages 466–481, Orlando, Florida, December 1987.

[SHO90] Stanley M. Sutton, Jr., Dennis Heimbigner, and Leon J. Osterweil. Language Constructs for Managing Change in Process-Centered Environments. In *Proc. of the Fourth ACM SIGSOFT Symposium on Practical Software Development Environments*, pages 206–217, 1990. Irvine, California.

[SK91] Michael Stonebraker and Greg Kemnitz. The POSTGRES Next Generation Database Management System. *Communications of the ACM*, 34(10):78–92, October 1991.

[SR86] Michael Stonebraker and Lawrence A. Rowe. The Design of POSTGRES. In *Proc. of the ACM SIGMOD International Conf. on the Management of Data*, pages 340 – 355, 1986.

[Sut90] Stanley M. Sutton, Jr. *APPL/A: A Prototype Language for Software-Process Programming.* PhD thesis, University of Colorado, August 1990.

[SZH+91] S. M. Sutton Jr., H. Ziv, D. Heimbigner, M. Maybee, L. J. Osterweil, X. Song, and H. E. Yessayan. Programming a Software Requirements Specification Process. In *Proceedings of the First International Conference on the Software Process*, Redondo Beach, CA, October 1991.

[TBC+88] Richard N. Taylor, Frank C. Belz, Lori A. Clarke, Leon J. Osterweil, Richard W. Selby, Jack C. Wileden, Alexander Wolf, and Michael Young. Foundations for the Arcadia Environment Architecture. In *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 1 – 13. ACM, November 1988.

[TW91] Craig Thompson and David Wells. Report on DARPA open OODB workshop I: Preliminary architecture workshop. Technical report, Information Technologies Laboratory, Computer Science Center, Texas Instruments Incorporated, May 1991.

[Uni83] United States Department of Defense. *Reference Manual for the Ada Programming Language*, 1983. ANSI/MIL-STD-1815A-1983.

[WWFT88] Jack C. Wileden, Alexander L. Wolf, Charles D. Fisher, and Peri L. Tarr. PGraphite: An Experiment in Persistent Typed Object Management. Arcadia Document UM-88-05, Software Development Laboratory, Computer and Information Science Department, University of Massachusetts, Amherst, Massachusetts, 1988.