(4)

AD-A214 478

RADC-TR-89-166
Final Technical Report
October 1989

THE CRONUS DISTRIBUTED DBMS PROJECT

BBN Systems Technologies Corporation

Richard Floyd, Sunil Sarin, Stephen Vinter

APPROVED FOR PUBLIC RELEASE; DISTRIBUTIION UNLIMITED

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

This report has been reviewed by the RADC Public Affairs Division (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-89-166 has been reviewed and is approved for publication.

APPROVED: Charles B. Schultz

CHARLES B. SCHULTZ
Project Engineer

APPROVED: Raymond P. Urtz Jr.

RAYMOND P. URTZ, JR.
Technical Director
Directorate of Command & Control

FOR THE COMMANDER: Igor D. Plonisch

IGOR G. PLONISCH
Directorate of Plans & Programs

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COTD) Griffiss AFB NY 13441-5700. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS N/A
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; Distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) 6973	5. MONITORING ORGANIZATION REPORT NUMBER(S) RADC-TR-89-166
6a. NAME OF PERFORMING ORGANIZATION BBN Systems Technologies Corporation 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Rome Air Development Center (COTD)
6c. ADDRESS (City, State, and ZIP Code) 10 Moulton Street Cambridge MA 02238	7b. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Rome Air Development Center 8b. OFFICE SYMBOL (If applicable) COTD	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F30602-86-C-0271
8c. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. 63728F; PROJECT NO. 2530; TASK NO 02; WORK UNIT ACCESSION NO 07

11. TITLE (Include Security Classification)
THE CRONUS DISTRIBUTED DBMS PROJECT

12. PERSONAL AUTHOR(S)
Richard Floyd, Sunil Sarin, Stephen Vinter

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Final	FROM Dec 86 TO Dec 88	October 1989	40

16. SUPPLEMENTARY NOTATION
N/A

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Distributed Operating Systems Distributed Database Management Systems Database Management Systems
12	07		

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Cronus is a distributed computing environment used to build and operate distributed applications. The distributed DBMS project has two main activities. The purpose of the Database Integration activity is to integrate a commercial, off-the-shelf database management system (DBMS) into Cronus, thereby making it available to any application using Cronus. The purpose of the Distributed DBMS activity is to conduct research into methods of infusing distributed database technology into Cronus. This report discusses the results of all work conducted under these activities.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☐ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT ☐ DTIC USERS	UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Charles B. Schultz	22b. TELEPHONE (Include Area Code) (315) 330-3623 22c. OFFICE SYMBOL RADC (COTD)

DD Form 1473, JUN 86 *Previous editions are obsolete.*

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

Table of Contents

FIGURES

1. INTRODUCTION

This document is the Final Report for the Cronus Distributed DBMS Project[1]. Cronus is a distributed computing environment used to build and operate distributed applications. The distributed DBMS project has two main activities. The purpose of the *Database Integration* activity is to integrate a commercial, off-the-shelf database management system (DBMS) into Cronus, thereby making it available to any application using Cronus. The purpose of the *Distributed DBMS* activity is to conduct research into methods of infusing distributed database technology into Cronus. This report discusses the results of all work conducted under these activities. The report has three objectives:

- to document in summary form the work completed during this project, identifying important results, lessons learned, and sources for more detailed information.
- to clarify the range of database options available to users and application developers within Cronus at the time of this writing (contact the authors of this report at BBN to determine the status of the current implementation); and
- to identify areas where further work will prove fruitful, both in developing additional extensions to the database facilities available within Cronus and in continuing database research in productive areas.

During the course of this project we considered a broad set of uses for database systems within the context of the Cronus environment. The breadth of the work was necessary to address two interrelated concerns: *near-term vs far-term availability of database systems*, and *the broad and frequently conflicting data storage and processing requirements of applications*.

In the near-term, we wanted to improve the database functions available to users and application developers within the Cronus environment. Toward this end, we developed software that directly addressed current needs, and also gained maximum leverage from commercially available database systems. In the far-term, we wanted to conduct research into several database-related areas to satisfy application requirements that were not being addressed by existing technology. Toward this end, we designed a distributed DBMS geared toward addressing the high availability requirements of our applications.

The database requirements of applications vary widely with respect to the structure of the data, the extensibility of data processing, the inherent distribution of the data, and the properties the data need to exhibit (e.g, high survivability or consistency). We addressed applications' various structural requirements by providing both relational and object-oriented data storage facilities. We addressed extensible data processing by allowing applications to incorporate data storage facilities directly into their application rather than requiring access to a stand alone data management facility. We addressed situations where data are inherently distributed by examining issues relating to federated databases. And we addressed application-specific data requirements by designing facilities to support access to

[1]Funded by the Rome Air Development Center contract F30603-86-C-0271, this project is a two-year collaborative effort between members of the Cronus Project at BBN and Xerox Advanced Information Technology. See Appendix A for the complete list of documents developed under this project, and references to Cronus-related documents.

replicated and distributed data.

Section 2 presents our work in integrating commercial relational database systems into Cronus. This work, described more fully in the Program Specification Report [Vinter 88b], has made it possible for Cronus applications to remotely access centralized databases. The two databases we have integrated are Informix and Oracle, though others (e.g., Ingres or Sybase) could be easily integrated, too. The strengths of these systems are the simplicity of the data modeling they offer (all data are structured as simple tables, or *relations*, of fields), the flexibility of dynamically manipulating the structure of the data, and the power of the query language they offer. All of these benefits are realized by remote Cronus clients, since the clients may exploit the full query language that these systems offer.

The primary weakness of the relational systems are their inability to customize data storage and processing to applications' needs, and their inability to satisfy application requirements relating to scalability and availability. Section 3 describes our work in extending the object-oriented storage facilities provided by Cronus to support distributed query processing. Cronus allows applications to embed data storage facilities directly in their application (thereby achieving customization of data storage and processing). Additionally, tools are provided to automatically support distribution of data among many hosts (achieving scalability) and replication of data (achieving survivability). These capabilities were extended by implementing an associative access query processing facility using Cronus objects, and detailed designs were developed to support distributed, multitype query processing.

The Functional Description [Vinter 88c] and System/Subsystem Specification [Sarin 88a] provide a detailed design of an advanced, distributed relational database system that supports highly survivable data management. These two reports provide a complete description of that work, and the reader is encouraged to obtain them. However, there were several research topics that arose from that design that are worthy of consideration, either in isolation or as part of a single system, for further research. These topics, notably directory management, query algebra, active monitoring, transaction management, distribution of functionality, and availability, are described in Section 4.

Under many circumstances data resides in several existing databases that cannot be moved to users (because of data in being collected or processed at remote sites) or converted to a single database system (because of cost and backward compatibility). Under these circumstances, neither access to a single centralized database nor access to application-specific data facilities are adequate to provide an integrated view of dispersed data. To address this problem, we examine issues and proposed solutions to provide *integrated access* multiple existing databases. This work is also described in Section 4.

In addition to the previously mentioned reports, the Technical Papers Report [Walker 88d] should be consulted by readers interested in the specific technical details of our work that are peripheral to the central work described in the other reports.

2. RELATIONAL DATABASE INTEGRATION

2.1. Overview

The goals of the database integration activity integration activity were to:

- evaluate and select a state-of-the-art, commercially available database product;
- provide host-independent access by any Cronus client to the selected database system;
- provide remote clients with the full functionality of the database system, including multistatement transactions and schema manipulation operations;
- support access control to the database system using Cronus client identities;
- reconcile the data model presented by Cronus, an object model, with the data model presented by the database system, a relational data model;
- support client/database interactions for the different types of uses databases commonly have (e.g., highly interactive vs. embedded);
- support queries over large databases that may require a long time to process and return a large amount of data.

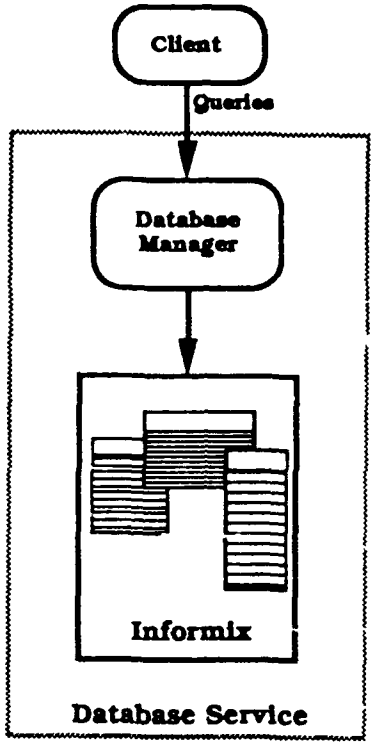
All of these goals were accomplished during the course of the project. Our evaluation rated Oracle, Ingres and Informix roughly equivalent in capabilities. We selected the Informix relational database system on the basis of lower cost and compatibility with other on-going RADC projects.

We implemented a Database Service that allowed clients fully functional remote access to Informix. The Database Service provides a basic building block for constructing vertical applications that rely on relational data storage facilities. Three such applications were implemented: a calendar system; a general-purpose forms interface for displaying the contents of any database in the DBMS; and a library catalog search system that provides on-line access to the BBN library database. We also implemented a prototype configuration service that uses Informix to maintain information on the hosts and services configured in a Cronus cluster.

The remainder this section provides a brief summary of the database integration work. A more detailed description may be found in other papers and reports produced during this project (see Appendix A).

2.2. Database Integration

The Informix database was integrated into Cronus by *encapsulating* it in a Cronus manager. The resultant Database Service acts as a front end through which all client requests are passed (Figure 1). This design allowed us to incorporate Informix with no changes to the Informix source (the source was unavailable in any case). The service provides fully functional remote access to the Informix database using Cronus object invocation mechanisms.



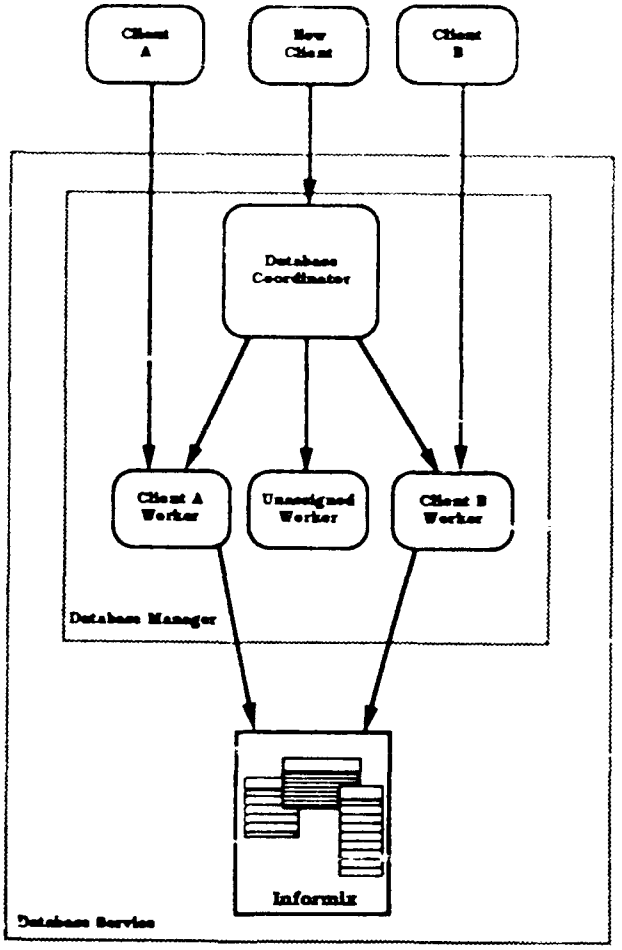
Database Integration Architecture
Figure 1

The encapsulation presented problems in the following areas:

1. Allowing concurrent access and update of the database;
2. Supporting operations whose formats and parameters can be composed by the client at run-time, not interface definition time, and whose results are determined using schema information contained in the database;
3. Providing access control restrictions, based on Cronus principals and groups, to restrict access at all appropriate levels of the database;
4. Supporting operations that may take an arbitrarily long amount of time, and that may

return large amounts of data.

The Database Service was actually implemented as multiple processes. This was done because of concurrency limitations in the underlying Informix DBMS. We developed a package, the *worker system* [Walker 88c], that transparently manages the multiple processes required in a fault tolerant fashion, allocating them to users as needed and freeing them when they are no longer in use (Figure 2). This is a general purpose package that has been designed to be easily used by other applications that require multiple processes.



Worker Architecture
Figure 2

An important difference between the Cronus object model and a relational database is the time at which the data types of the results of an operation are determined. Cronus conventionally defines parameter data types at interface definition time. This strong typing results in well-defined interfaces that greatly ease the construction of large systems. However, it is not appropriate for interactive use of a relational database, where the contents of a request and its result are specified at invocation time. We added a *self-describing type* to Cronus to support relational DBMS operations, such as Select, that return results of varying types.

The Database Service provides access control, based on Cronus principals and groups, at the database, table, column, and view level. Informix maintains protections based on local host user names. The Database Service reconciles these differing models by maintaining its own Cronus-based protection information at each level in tables stored in Informix, and using this information to validate a query before it is executed.

Database operations may potentially return a large amount of data, with a correspondingly large transmission overhead and delay. We implemented an option that allows clients to specify a limit on the amount of data returned in a Select, and to retrieve more data as needed.

Our initial attempts at using the Database Service were frustrated by the use of timeouts in Cronus to detect host and service failures. Unfortunately, these timeouts were short enough that they were triggered by any Informix operation that processed a non-trivial amount of data. We addressed this problem by designing and implementing an extension to the Cronus invoke mechanism that eliminated most timeouts from object invocation [Bono 88, Floyd 88]. The general approach taken was to make invoke requests themselves objects (in a "lightweight" sense), and to allow clients to use the standard Cronus locate mechanism to verify at any time that a request was still in progress.

The database integration work that we have described here has been used in several other applications. As part of a separate BBN project to provide an expert system for assessing naval capabilities, we developed a Database Service for the Oracle relational database system. This service is virtually identical to its Informix counterpart, and we anticipate that a version to support Ingres, Sybase, or any other SQL-based relational database system could be developed in several man-weeks.

We have also implemented a prototype configuration service. This service uses Informix to maintain information on the hosts and services configured in a Cronus cluster, but presents to users an object-oriented view of the data that hides the SQL query language used by Informix. This is an example of an embedded interface to a database, and is appropriate when the service is part of a larger system where the need for a well defined and static interface is paramount. This style of use would also appropriate when processing must be done on the data before it is returned to the user, or if an operation requires invocations on other Cronus objects.

2.3. Specific Uses

We developed three applications that make use of the Informix Database Service: an electronic calendar, a library catalog search system; and an interactive forms interface. This has given us considerable experience in implementing and using applications that are based on the Database Service.

The electronic calendar [Walker 88f] is used to keep track of BBN personnel activities, including travel, vacations, visitors, and events (meetings, seminars, demonstrations, etc.). People can examine the calendar for any selected period and for any group of people (e.g., by project, department, or selected individuals). Several different interfaces are provided to users for accessing the calendar. For users with Cronus available on their machines, a forms-based calendar interface can be used to display schedules in three modes: as a graph, as a table, or within a form. Additionally, sophisticated users may submit SQL statements. Users who do not have Cronus executing on their machine may interact with the calendar through mail. Users may also register themselves to receive a mail message automatically each morning containing the day's calendar for a selected group of individuals (e.g., Cronus project members). Additionally, the Service can be used to generate a hardcopy printout of the calendar for the upcoming three week period. This application is in daily use at BBN in several departments.

The Calendar Service provides regular use of the database service software, but its usage patterns rarely cause greater than two concurrent accesses at any point in time. To test and evaluate the Database Service under operational, heavy load conditions, we developed a second application, the Library Catalog Search System [Walker 88a]. This application provides on-line access via Cronus to the BBN library database. The application was developed in approximately two weeks, demonstrating the ease of use of the Database Service under Cronus. We anticipate that it will receive use by hundreds of BBN users, thereby providing an excellent test of the software under heavy load conditions.

We also developed an interactive, general purpose forms interface to the Database Service. This interface, CRSQL [Walker 88b], gives an interactive user the ability to generate and modify database tables using a form driven interface. In addition, users can create and drop databases, examine schema, update access control information, and interactively construct and execute SQL statements.

2.4. Future Directions

Integration of other commercial off-the-shelf database products would provide new platforms for application development and expand the user base. Integration of new DBMSs is complicated by differences in SQL syntax, access control, and programming language interfaces between various products. However, our experience with Oracle suggests that these difficulties may be overcome with a relatively modest amount of effort given our current implementation support.

The model used by Cronus assumes that a single process manages all objects of a given type on a host. This assumption is not warranted if an embedded application makes use of the worker system. Relaxing this restriction, and resolving the resultant security and generic object invocation issues, would ease the use of worker processes in embedded applications.

The CRSQL interface was developed to demonstrate the capabilities of the Database Service. We did not commit the effort needed to provide extensive forms capabilities. For example, it does not generate reports data input is relatively awkward, and the forms it creates are fairly simple. A considerable amount of effort could be placed in improving the interface.

Allowing clients to store queries for later execution would significantly improve the usability of the database. The only means provided currently for re-executing queries is to read them from files and resubmit them. Informix provides no means of storing queries for later execution. However, this feature could be fairly easily added to the Database Service.

Cronus provides support for the replication, distribution, and reconfiguration of objects. By providing a centralized Database Service, we lose the potential benefits of these capabilities. It may be beneficial to investigate methods for supporting replicated and distributed data contained in commercial off-the-shelf databases in the context of Cronus.

3. CRONUS QUERY PROCESSING

3.1. Introduction

In the previous section we described the integration of a commercial off-the-shelf relational database into Cronus. This provides Cronus users and applications with a service that allows data to be stored as relations and accessed associatively. However, one of the strengths of Cronus is its object-oriented architecture. Cronus includes facilities that allow application data to be embedded in the application. Unlike data contained in a centralized relational database, these data may be structured as objects, replicated, and distributed based on the needs of the application. These capabilities are currently supported by Cronus.

There had long been a need for associative access to Cronus objects, but existing applications generally didn't attempt to provide such access because of the complexity involved in doing so. Applications (such as the Cronus Bug Reporting Service) that did support associative access did so in very limited ways. It was necessary to modify a manager for each new type of query, and the distribution and replication of objects was not allowed to insure correct results. This situation led us to investigate providing associative access to Cronus object data.

Cronus objects of a given type may be managed by a number of managers distributed across the network. In addition, Cronus allows users to define new object types. Because of this, the problem of providing associative access to Cronus objects breaks down naturally as follows:

1. Query processing on objects of a single type in a single manager;
2. Distributed query processing on all objects of a given type;
3. Query processing across multiple types.

We discuss the work we have done in each of these areas in the following subsections.

3.2. Integrating Query Processing into a Cronus Manager

Our primary goals in implementing query processing on Cronus objects were that the result be easily integrated into existing and new Cronus services, that it be sufficiently flexible to meet a wide variety of needs, and that it provide a convincing demonstration of object-based query processing in Cronus. We met these goals through the use of type inheritance, a flexible SQL-like query specification language, and integration of the developed facilities into an existing Cronus service.

Cronus allows operations defined at higher levels of the type hierarchy tree to be inherited by lower levels. We have taken advantage of this type inheritance to define a query type that supports general query operations that can be inherited by other types. A detailed description of the interface to this type is described elsewhere [Walker 88c]. We present a brief summary here.

The query type defines four new generic operations: *Query*, *AddIndex*, *DropIndex*, and *ShowSchema*. The *Query* operation allows users to perform a query on all objects of the invoked type in a manager. It takes as a parameter an SQL-like query specification that includes a description of the fields to query, the relationships between fields that must be satisfied for an object to match the query, and so on. The result of a query operation is the list of UIDS matching the query. The user can optionally ask that selected fields from each matching object also be returned and that objects be sorted by a specified field.

The operations *AddIndex* and *DropIndex* allow indices for a type to be dynamically created and deleted. Indices are used to speed up evaluation of a query by reducing references to the object database. Without indices, performing a query requires a scan of the entire object database of a type. When a query is received, it is reordered so that fields that are indexed are evaluated first. Indices are used to select objects that match the query, and then, if necessary, the object database is consulted to finish the query.

The *ShowSchema* operation describes the fields that may be referenced by a query, and shows which fields are indexed. The fields that may be queried are defined by the implementor of the type manager. They may be scalar or array fields of the object being queried. In addition, the developer may define *computed fields* that are calculated at query execution time using a implementor-supplied procedure.

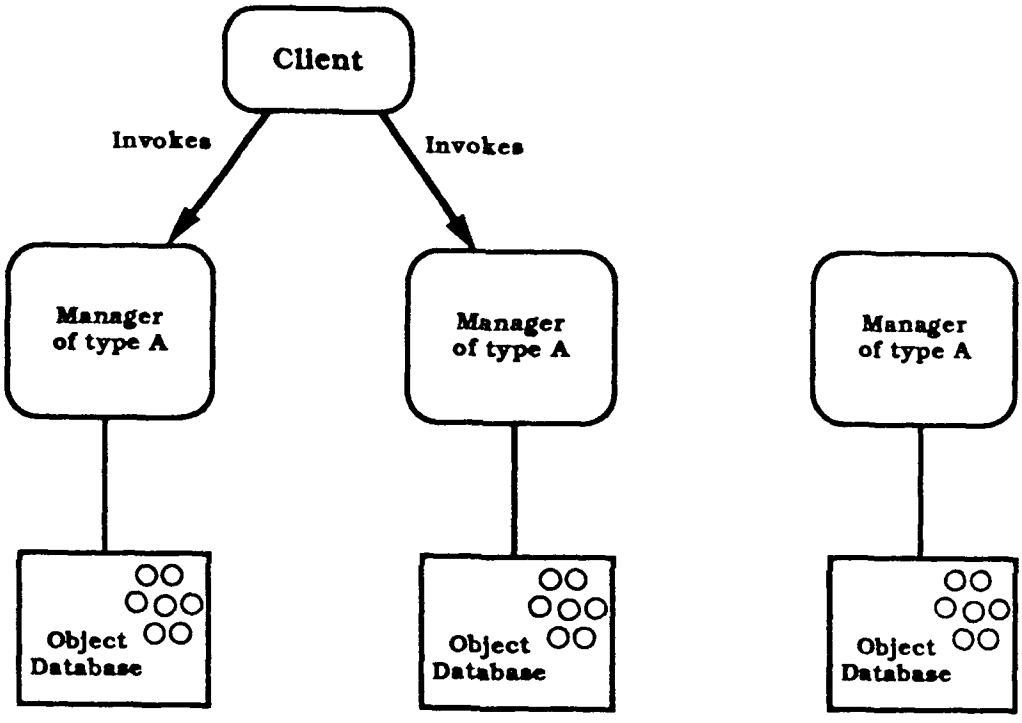
To incorporate the query processing facility, the developer of a manager for a type makes his type a subtype of the query type. He then defines and names the fields that may be referenced in a query (this is done to allow the actual implementation of objects to remain hidden), and defines functions for any computed fields. Finally, he rebuilds his manager to incorporate code for the query facility.

Using these facilities, the developer of a manager can present to the user the abstraction of a relation that may be associatively queried, using an SQL-like language, in a manner similar to a relation in a relational database. However, in this case the columns of the relation are made up of fields of Cronus objects and computed values, and changes to objects are reflected in the relation seen by the user.

We developed our query processing implementation in the context of the Cronus Bug Reporting Service, although the design will allow it to be integrated into any manager. We chose the bug reporting service because of its need for complicated queries on bug reports containing fields of various types. The ad-hoc nature of the partial query implementation used by the existing service made it difficult to extend, and it was frustratingly slow. We were able to completely replace the existing query processing facilities of the old service with our implementation, resulting in much simpler service that will be easy to extend and enhance in the future. One gratifying result of this integration was the speedup achieved through the use of indices. By adding indices to commonly used bug fields (such as keyword and the date the bug was reported), we were able achieve better than an order of magnitude speedup for typical queries. The query-based bug report manager is now in everyday use at BBN, and the interactive interfaces that use it have been converted to use the new query facility.

3.3. Distributed Query Processing in Cronus

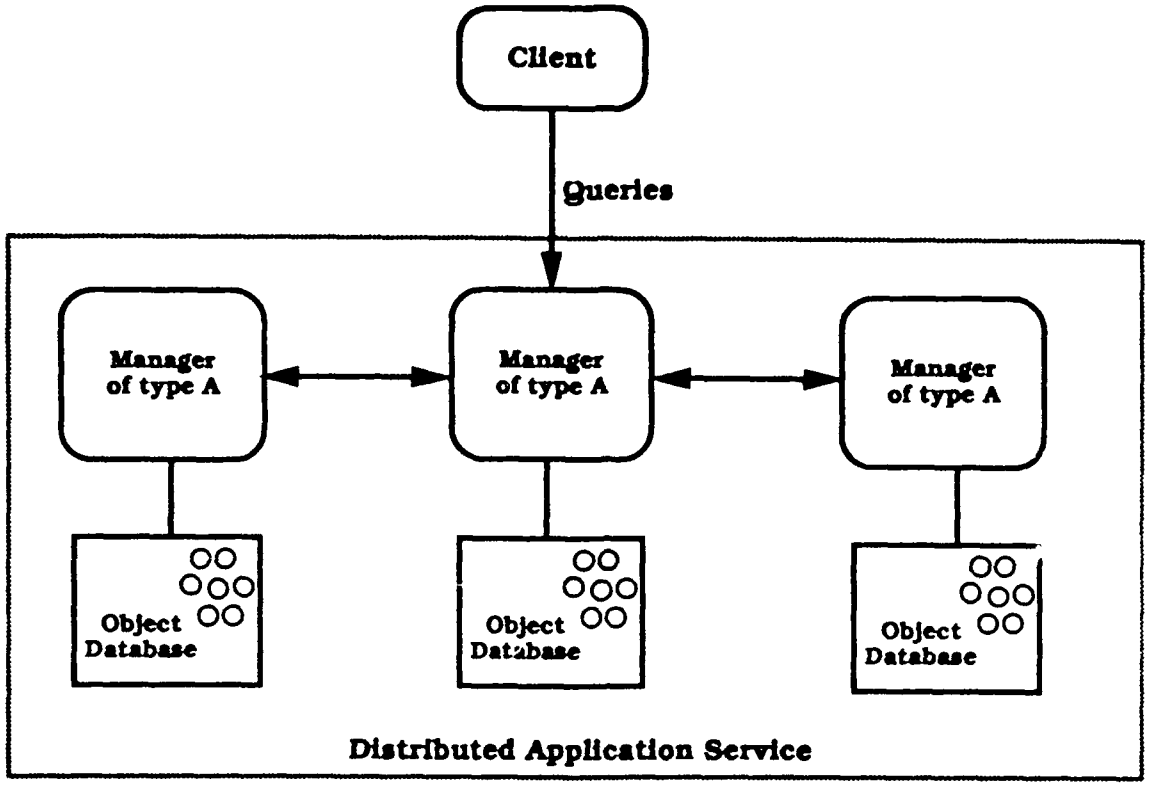
A Cronus type may be supported by a number of managers scattered throughout the network (Figure 3). Supporting a coherent view of the objects of such a type requires running a distributed query across these managers. We have done extensive design work on this problem [Vinter 88a].



Accessing Distributed Objects of a Type Using Invokes
Figure 3

If objects are unreplicated, it is necessary to contact all managers of a type to insure that all objects of the type are queried. Each manager maintains information on the location and status of other managers of the same type. When a manager receives a query, this information may be used to forward the query to other managers of the type. In our design, the manager that receives the initial query (the *coordinator*) does this, collects the results from other managers, and then returns the entire set of objects of the type matching the query to the user (Figure 4). This makes the actual distribution of the objects of a type transparent to the user.

Objects in Cronus may be replicated to increase their availability. A combination of weighted voting [Gifford 79] and version vectors [Parker 82] is used for this. Because of the use of voting, a given copy of an object may not be up to date. If a consistent view of the data is desired, then enough information must be collected to insure that the result is based on current data (our design also



Distributed Query Processing in Cronus
Figure 4

supports *quick queries* that provide a generally correct view of the data at a lower cost).

The problem of querying replicated objects, then, has two parts: 1) determining the latest version of the objects; and 2) evaluating the query predicate on objects to see if they satisfy the query. These two steps may be done in either order. In addition, the execution of these steps on objects may be coordinated either from a central site (the site that receives the query), or by a site that actually manage copies of objects being queried. Based on these considerations we designed four query processing strategies:

1. CC-PF: Predicate evaluation first with centralized coordination;
2. DC-PF: Predicate evaluation first, with decentralized coordination;
3. CC-LVF: Find latest version first, with centralized coordination;
4. DC-LVF: Latest version first, with decentralized coordination.

We evaluated these strategies based on the cost of network transfers, messages required, the computational cost, parallelism allowed, and scalability for a number of different configurations. The evaluation pointed out factors that are important to consider in developing query processing strategies for replicated, object-oriented data, and the impact of those factors on the cost and optimality of the strategies. Our evaluation led to adopting the DC-PF strategy for eventual inclusion in Cronus. DC-PF is an optimistic, decentralized strategy that scales well, uses smaller messages, and offers increases parallelism over centralized approaches, although it does so at the expense of an increase in the number of messages required. The increased message cost is justified in our environment, given the benefits of the approach.

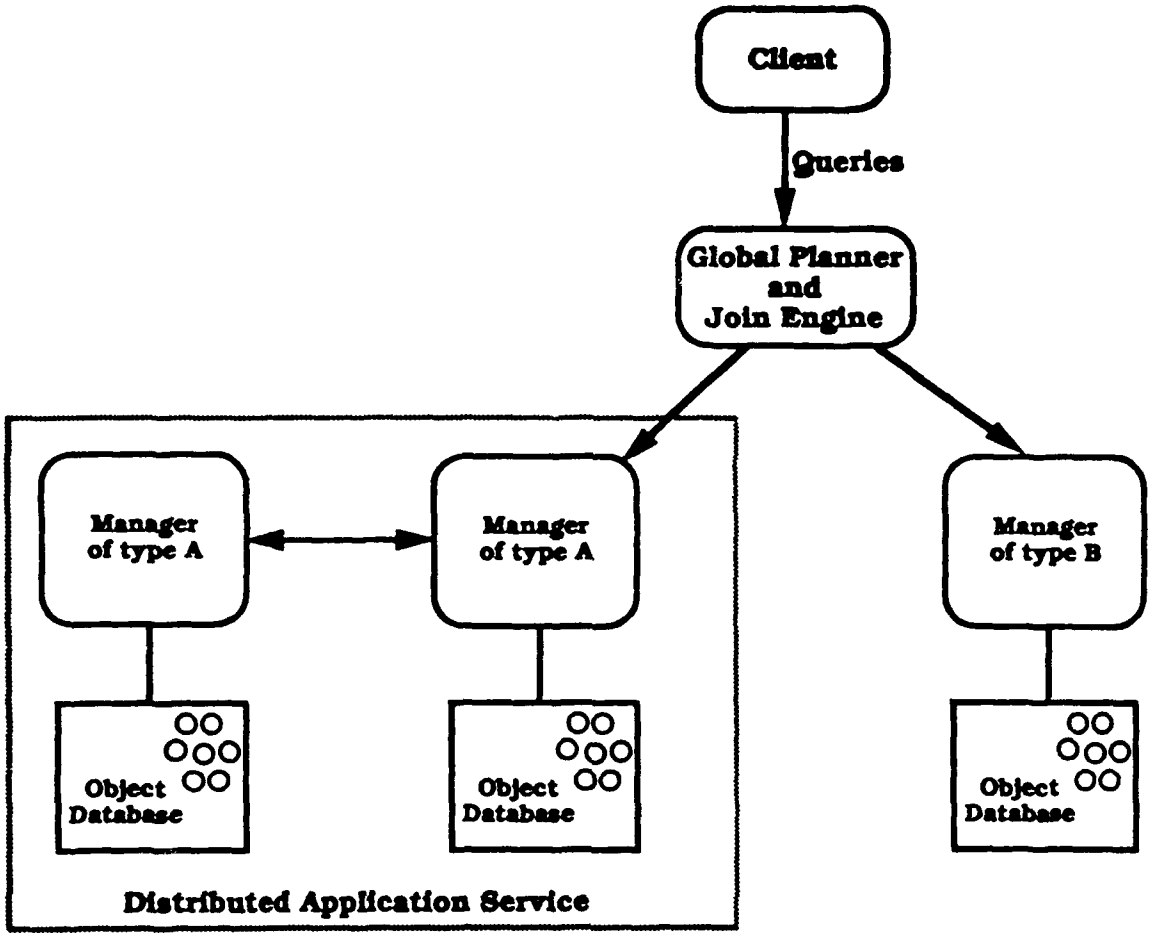
3.4. Multitype Query Processing in Cronus

A relational database allows queries to be performed over multiple relations. If the objects of a Cronus type are viewed as supporting a relation, the corresponding capability in Cronus is the ability to perform a query that spans multiple object types. We have done extensive design work on this multitype query problem in the Cronus environment [Phadnis 88].

Our multitype query work builds on the distributed query processing design we have described in sections 3.2 and 3.3. We view each type as supporting a relation, with the relation being derived using the techniques described in the previous sections. The multitype query problem can then be reduced to a select-project-join query. That is, we use the DC-PF algorithm on each type to select objects of the type that match the applicable portions of the query, project to obtain the desired tuples, and then join the resulting relations to arrive at the result.

We have identified two additional functional components that are needed to perform multitype queries. The *Join Engine* coordinates the transfer of data from coordinators for the types involved and actually performs the join. The additional complexity involved in performing queries across multiple types motivates the inclusion of a *Global Planner*, which decomposes queries and optimizes them to reduce execution cost. We analyzed several architectures for performing multitype queries in the context of Cronus and our previous distributed query work. An architecture that implements the join engine and global planner together, but separates them from the individual type coordinators, was found to be the best choice given our goals and environment (Figure 5).

Joining two projected relations at what may potentially be a third site can result in substantial network traffic. Our design includes an optimization technique, *semijoin-based reduction*, that reduces this overhead by pre-filtering a projected relation to eliminate tuples that can't appear in the final join result.



Multitype Query Processing
Figure 5

3.5. Future Directions

The current Cronus object database implementation is not well matched to query processing. In particular, adding the ability to scan objects in physical (as opposed to random) order could be expected to result in substantial performance improvements.

Our query processing implementation was designed for small to moderate-sized (1-20,000 records) object databases. Supporting very large object databases will require additional work both on indexing strategies and on the Cronus object database itself.

The current query processing design and implementation do not allow for updates on an associative basis (for example "delete all bug objects belonging to rfloyd that are not cataloged"). Such updates can be mimicked by a combination of a query followed by individual operations on the returned objects, but a more integrated approach would have performance and ease of use benefits.

We implemented an interactive interface to the database service (CRSQL) that allowed arbitrary SQL queries to be performed on relations in the database. A similar interface to managers supporting queries would enhance the usability of the query facility and of the managers supporting it.

Our implementation work has demonstrated the benefits of associative access to Cronus objects. However, the implementation has been limited to queries on a single type and manager. Extending the implementation to distributed objects and multiple types, as described above, would significantly enhance the associative access capabilities available to applications in Cronus.

4. DISTRIBUTED DBMS DESIGN FOR CRONUS

The design of a distributed DBMS capability for Cronus was the major research task of this project. The primary results of this design effort are documented in the Functional Description [Vinter 88c] and System/Subsystem Specification [Sarin 88a]. The goals of the design were that the proposed DDBMS exhibit the following properties:

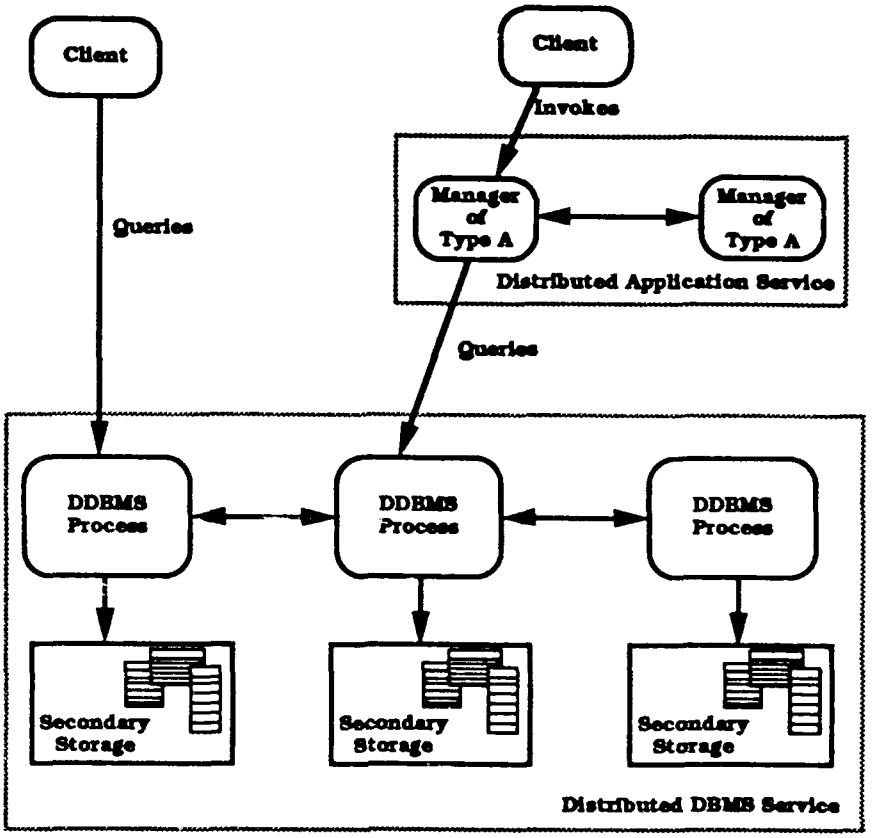
- Integration with the Cronus model and software development environment.
- A global view of data, that hides network structure and machine boundaries.
- High availability and survivability, in the presence of failures of large numbers of hosts or communication lines.
- Scalability, to very large numbers of hosts.
- Reconfigurability, permitting dynamic reallocation of data in response to failures or changing load patterns.
- Extensibility, permitting more powerful capabilities to be added in the areas of data modeling and manipulation, monitoring of constraints and triggers, and automatic reconfiguration.

A distributed DBMS with these properties would be sufficiently flexible and survivable to provide a database service supporting a wide range of Cronus client and application needs (Figure 6).

The results of the design effort are summarized below. Section 4.1 overviews the functional capabilities that we proposed for the Cronus DDBMS, and the suggested implementation approaches to these. Section 4.2 discusses system architectures for tying together all of these capabilities, and examines the important tradeoff between integrating existing database management software and developing new software that is more tailored to the distributed capabilities envisioned. Section 4.3 then suggests directions for further pursuing this work.

4.1. DDBMS Functionality

We designed a distributed DBMS interface that incorporates a variety of advanced capabilities not available in today's database management systems. For each functional area below, we present our model of how the functions should be perceived and used by clients of a database, and then briefly discuss available implementation techniques for realizing this functionality.



Distributed DBMS Service
Figure 6

4.1.1. Data Manipulation Capabilities

We have proposed the relational model as the basis for interacting with a database, because the high-level view it presents insulates clients from the details of the physical representation of data. The relational model appears flexible enough to accommodate many features found in recent "semantic" and "object-oriented" data models. The Probe project at XAIT [Orenstein 87], for example, allows for "virtual" relations that contain no stored data but are realized by a body of programming language code. We have retained this concept of virtual relation in our proposed design.

We also developed an algebra of operators for querying and updating a database. Our intention here was to provide a computational model that is powerful and extensible and amenable to the transformations and manipulations that must be performed within a distributed DBMS. The query algebra, based on work done in the Probe project, allows any relational expression to appear as an operand to any relational operator, and also includes more powerful operators for computing outer-joins and aggregate functions. Ultimately, external interface languages will need to be supported, such as SQL or a forms-based or graphical interface; all of these should be translated into the internal algebra for processing by the system.

Our update operators (for insertion, deletion, and modification of tuples) are again algebraic in form, and operate associatively on sets of tuples. The approach allows for many useful kinds of updates that cannot be easily expressed in languages such as SQL. Stored update procedures, which may include conditional statements and expressions, are also supported. Iteration facilities are built in; a given procedure may be invoked on any relational expression, and will execute the procedure body on every tuple in the argument relation.

The implementation of the proposed query and update capabilities requires translation of high-level requests into sequences of operations on data stored at possibly different sites. We used the algebraic approach again to define a collection of "physical" operators on streams and stored sets of tuples, which takes into account properties of the streams and stored sets such as sort order, presence of indexes, and physical location. Transformation rules specify when two algebraic expressions are equivalent, and optimization rules specify when it is profitable to transform an expression into another equivalent one. Rule-based optimization of queries [Rosenthal 86] allows for extension in both the set of operators and in the set of rules, and is proposed as the appropriate technique for implementing the distributed data manipulation capabilities.

4.1.2. Active Data Management

This functional area includes the following capabilities:

- Materialization of views, that is caching a copy of the result of evaluating a view definition. This can improve query performance if the view is accessed frequently relative to the frequency of updating the base data; this includes any accesses to the view that are needed in order to monitor constraints and exception conditions (below). In a distributed system, materialized views can provide "snapshots" of data that are located at remote sites [Adiba 80] and that may not be available at the time of access.
- Monitoring constraints and exception conditions on the data. The mechanism we propose for doing this is to define an appropriate view and to attach properties such as "prevent-adds" or "report-deletes" to the view definition. These properties specify that the addition (or deletion) of a tuple to the view should either be disallowed or that it be permitted but the update reported. The latter option is useful for "soft" consistency constraints, and also for cases where a client program has cached the results of a query in order to perform some lengthy computation (or to allow an interactive user to browse over the data), and would like to be notified when its cache becomes out of date.
- Triggering of actions based on conditions on the data. This is done by attaching side-effect specifications to base relations or appropriately-defined views; these side-effects specify additional tests or actions to be invoked when tuples are added to or deleted from the given relation or view.

Implementation techniques for the above are being developed in many "active database" research projects, e.g., HiPAC [Dayal 88] and Postgres [Stonebraker 86]. Although we expect to use these techniques, they have been developed for centralized systems and need extension in the distributed environment. In particular, it will be necessary for the distributed DBMS to keep track of which sites are holding copies of which views, so that updates to base data can be propagated to sites that need to know this information.

4.1.3. Transaction and Session Management

The conventional model of database interaction allows a client to submit a sequence of query and update requests that are executed and committed (or aborted) as a single serializable transaction. Such potentially long transactions have serious effects on availability because the client's transaction can lock out other transactions for long periods of time; this is especially severe in a distributed system. Our design tries to get around this problem by minimizing the need for multi-statement transactions. That is, the data manipulation capabilities allow a client to pack a significant amount of computation (including multiple queries, updates, as well as conditional control) within a database update request. By placing many intermediate computations in the database manager rather than in the client, the duration of a transaction is reduced (reducing the adverse effects on availability) and so is the amount of data transferred between the database manager and the client.

Multi-statement transactions on a database are still be supported, for cases where the above is not sufficient for the client. These should be implemented by an "optimistic" concurrency control protocol that does not lock out other concurrent transactions. Unfortunately, the longer such a transaction lasts, the more likely it is to abort when an attempt is made to validate its serializability and commit it. The client may choose to relax some serializability and consistency requirements for long transactions, in order to reduce the probability of abort and the amount of lost work. If data has been read during a transaction, the client can request that commitment of the transaction not be dependent on these data remaining current for the duration of the transaction. Instead, the client can use the active monitoring capabilities described above to receive notification of changes to data read, and decide how to react; depending on how serious the change is, the client may or may not wish to abort its ongoing transaction.

If a transaction consists of more than one update request, the client may request that earlier updates be committed (by validating and forcing to stable storage) even if subsequent ones are unable to pass the validation test and commit. The client may also ask that the system make its "best efforts" to serialize a sequence of updates atomically, but that the updates be executed anyway, as separate atomic actions when this is not possible. This is related to the topic of the next subsection.

4.1.4. Trading Consistency and Availability

The previous subsection discussed treatment of transactions consisting of multiple database requests. The issue of consistency and availability can arise in the execution of even a single database update request; this may involve operations on multiple fragments that are distributed, or may involve checking constraints as well as performing actual updates, or may involve accessing and updating multiple copies of the same data. Performing such an update request as a distributed atomic transaction ensures consistency but is vulnerable to extreme communication failures; the concurrency control and recovery protocols will block (and prevent the execution of further transactions) if there is a network partition. This cost may be acceptable in some applications, but in most military environments it is critical that database operation continue in the face of severe damage to sites and communication lines.

Our approach to trading consistency and availability is based on adapting and extending SHARD [Sarin 85], the System for Highly Available Replicated Data that was developed by XAIT for the Strategic C3 Experiment. SHARD took an extreme approach to availability, by allowing any site to continue operation (including updating the database) even when completely isolated from other sites; possible inconsistencies are resolved when they are detected, i.e., when sites that were partitioned can communicate again. SHARD also assumes full replication of the database at all sites, which is too restrictive for large systems.

The more flexible approach that we propose is to have transactions behave serializably during "normal" conditions when sufficient sites can communicate, allowing inconsistencies to occur only if there are communication problems. A client-submitted database request is treated as a "compound transaction" that may be broken up by the system into smaller atomic actions, each typically operating on a different site. An attempt is made to execute the entire compound transaction atomically, but if this is not possible within a client-specified timeout period (an idea first proposed for SHARD [Sarin 86]), the transaction is executed as separate atomic actions. Queueing and retry mechanisms are also provided to ensure eventual execution of each atomic action, in case not all of the sites are available at the same time.

The mechanism for trading consistency and availability is intimately related to the active data management capabilities. First, the actions that comprise an update transaction must include all necessary checking of constraints. Because of communication failures, the entire transaction may not be atomic, which means that some constraint checks may not be part of the same serializable action as the actual database update; as a result, a constraint may appear to be satisfied but may turn out to be violated when communications are restored. Active data management capabilities are important here as well, in that the system may automatically trigger compensating actions (to automatically rectify the problem, or inform an appropriate user) when a constraint violation is detected.

4.1.5. Directory Management

Each relation (or other named entity, e.g., view or stored procedure) in a database has a fair amount of directory information associated with it: the logical schema (column names and types, active monitoring properties), distribution information (where copies are stored), physical storage information (how the storage of a particular copy is organized and indexed), statistical information (load, cost, and performance parameters), and administrative information (access control, documentation). The directory information is needed to allow proper translation of high-level requests into sequences of operations on stored data. It is desirable that directory information be highly available, in particular that the directory information for a given relation be at least as available as the data comprising the relation itself.

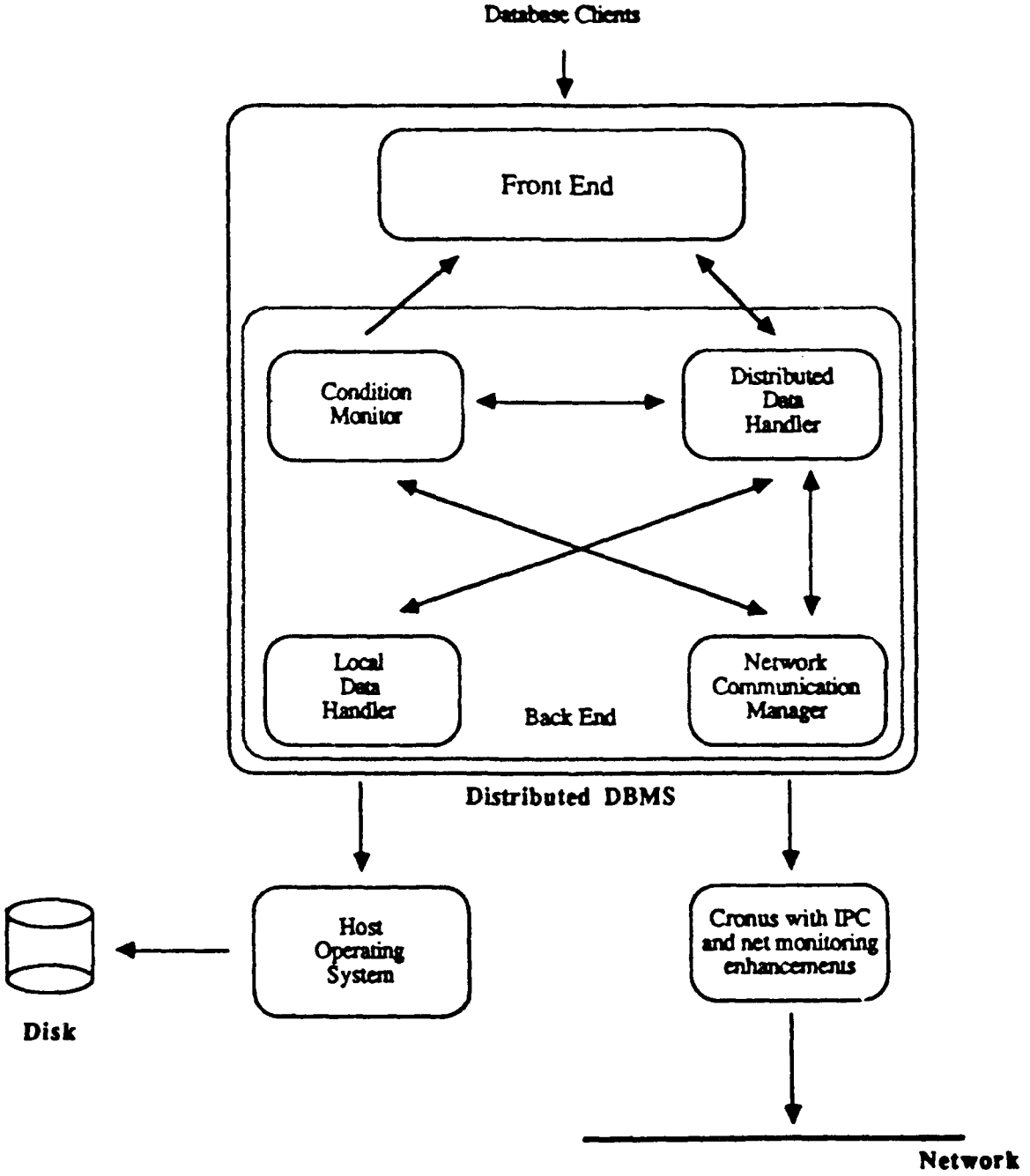
We surveyed a variety of directory management techniques and found that most did not have the above property. For the most part, all names within a given directory (or subdirectory) have their directory information replicated at the same set of sites. However, individual relations must be permitted to migrate to arbitrary sets of sites (this is essential for scaling and load balancing). It is therefore possible for copies of a relation to be accessible while sufficient copies of its directory information are not, unnaturally blocking access to the relation.

To solve the above problem, we developed a more flexible directory replication algorithm [Sarin 88c]. Rather than having the same set of sites control an entire directory, the algorithm allows each name to potentially have a different set of controlling sites; it is easy to make this set be the same as the set of sites controlling the stored relation associated with the given name. Ranges of unused and deleted names are managed as "gaps" (an idea borrowed from the algorithm of [Bloch 87]) and each gap has an associated set of controlling sites. Insertion of a new name into the directory requires permission from a quorum of the sites controlling the gap into which the name falls. Control over the name can subsequently migrate to other sites, and the directory lookup protocol uses version numbers to determine which sites hold current directory information for a given name.

The directory replication algorithm uses weighted voting to ensure consistency and serializability. Further work is needed to make the directory information accessible and updatable during network partitions, by incorporating, for example, Cronus's "version vector" approach to replication.

4.2. DDBMS Architecture

The distributed DBMS architecture must tie together the solutions proposed for all the functional areas described above. The desired architecture differs from known conventional DDBMS architectures (such as DDM [Chan 87] and R* [Lindsay 84]), because of new capabilities such as active condition monitoring, and because of a different approach to availability and consistency. We proposed [Sarin 88a, section 3] an architecture with five main components (Figure 7):



Distributed DBMS Component Architecture
Figure 7

1. A Front-End (FE), which provides host language and interactive interfaces to database clients, performs some initial parsing and name resolution for submitted queries and transactions, and formats results for presentation to the client.

2. A Distributed Data Handler (DDH), which performs validation, access control, planning, and optimization for submitted requests, and also coordinates their execution and performs concurrency control and distributed commit and abort processing. The DDH manages the directory information that is used by query planning and that is updated by reconfiguration commands and by statistics-gathering activities; because directory information is partially replicated (using the algorithm discussed above), no site's DDH is required to maintain complete directory information.

3. A Local Data Handler (LDH) that manages stored data. In the course of its participation in a global query plan, the LDH performs some simple local query optimizations and executes its portion of the plan against the stored physical data structures. The LDH also performs local lock and log management and statistics collection.

4. A Network Communication Manager (NCM) which provides basic inter-process communication primitives, including multicast communication for replicated data and for commit processing. The NCM monitors the status of hosts and communication paths, providing an abstract model of host status and connectivity that is used by the DDH in query planning.

5. A Condition Monitor (CM), which receives as input notifications of changes to data and to network status, and tests exception conditions specified by clients in order to notify them (or trigger actions automatically, e.g., compensation for inconsistent updates during a partition) when a condition occurs.

Except for the CM, which represents the major new functionality in our proposed system, the components reflect the architectures of most conventional distributed database systems. A major issue in defining the above components is the nature of the interface between global data management (DDH) and local data management (LDH). We discuss below two main architectural approaches to this, one for an classical "top-down" distributed DBMS where large parts of local data management functionality can be designed from scratch, and one for a "bottom-up" distributed database system that provides integrated access to multiple existing databases. We also briefly discuss the issue of operating system support for distributed DBMS implementation.

4.2.1. Migrating Functionality into Local Data Handlers

Achieving full distributed DBMS performance and functionality requires from local data handlers some capabilities not found in a conventional centralized DBMS. These capabilities include participation in two-phase commit (for distributed atomic transactions), access to log information (for incremental recovery of replicated data), a query interface at the physical access method level (allowing global distributed query optimization), and the ability to accept a data stream or send a data stream to a process other than the one that issued a query (also for distributed query optimization). These

functions are provided by the local data management components of most homogeneous distributed DBMSes (such as DDM and R*), and we continue to assume that such functionality should be incorporated in the LDH.

Our original architecture assumed that condition monitoring (CM) is mostly a global function that interacts with the DDH. Again, however, we believe that achieving reasonable performance will require some support for condition monitoring at the local level. The LDH can notice an exception condition fairly quickly if all the data needed to test the condition is materialized at the local site. Thus, an LDH that performs condition monitoring (the HiPAC system under development at XAIT is an example of this) may be appropriate if combined with a mechanism in the DDH that propagates updates to all affected copies. Further research is needed to determine the best partitioning of condition monitoring functionality between the global and local levels.

4.2.2. Integrating Existing Databases

Designing an LDH from scratch may be the path to better DDBMS performance, but many organizations have existing databases and applications in which they have significant investment. Furthermore, even for data that is not already in a database system, purchasing an off-the-shelf DBMS is cheaper and provides at least partial functionality sooner than developing a new DDBMS.

We have addressed the problem of multiple database access in [Sarin 88b]. While the use of existing database software imposes some constraints on distributed DBMS capabilities (e.g., in distributed query and transaction management), we believe that a reasonable amount of functionality can be achieved with some simple mechanisms that would not require a major research and development effort:

- Support for "snapshots" allows a database manager to cache information that is derived from other databases. This is a useful technique when availability is important and instantaneous currency of the answer to a query is not critical. Its implementation avoids many of the heterogeneity problems associated with processing queries over multiple existing databases.
- Data in existing off-the-shelf databases can be integrated into the proposed Cronus distributed query system [Phadnis 88] by developing a "gateway" that translates (a subset of) the Cronus query language into (a subset of) the vendor's query language. Limited optimization will be performed for queries involving such data, but this can coexist with the query optimizations that we have proposed for queries across multiple types in Cronus.
- Because of the inability to modify existing DBMS software, update transactions to multiple databases should use the techniques discussed above for achieving high availability, namely committing individual subtransactions separately, performing compensation, and queueing subtransactions for eventual execution.

Ultimately, we believe that both existing local databases and newer local database software tailored for distributed data management (developed under contract, or by DBMS vendors who are introducing distributed capabilities) will need to coexist. In order for this to be feasible, it will be necessary to define multiple levels (say logical query language and physical access method) of interface to local data managers. A given local data manager may then support any one or more of these interfaces, and this will be taken into consideration by components (the DDH) that perform global query planning.

4.2.3. Operating System Support for the Distributed DBMS

We investigated [Vinter 88c, section 3]) the extent to which Cronus provides operating system support for building the proposed distributed DBMS. For local storage management, the ability to force buffered pages to disk is the most critical function that needs to be added (in order to properly support transaction recovery); more sophisticated functions such as buffer management and raw disk access are less urgent and can be considered later if performance appears to be a problem. The existing Cronus IPC is largely sufficient for most communication requirements, although a multicast abstraction would be convenient. In addition, monitoring of host and network status is important and would be more efficient if it were implemented in Cronus than if it were placed in the DDBMS software.

4.3. Future Directions

Developing a DDBMS with all of the features we propose is an ambitious task. For the short-term, subsets of this functionality can be implemented and made available in Cronus, e.g., simple "snapshot" access to multiple off-the-shelf databases or "gateways" from the Cronus query language to off-the-shelf databases. In the long run, however, the shortcomings of current DBMS technology will be felt by demanding applications. These include the lack of extensibility and computational power (as exhibited by today's SQL) and the lack of active notification and triggering facilities. Some of these shortcomings are being addressed by DBMS vendors and it is not unreasonable to expect products in the next five years to exhibit many of these features. There is one important difference, however, in that high availability and survivability are less critical in a business environment than in military applications such as command and control. Therefore, developing and refining techniques for trading consistency and availability will require a concentrated research effort since it may not be covered adequately in the DBMS marketplace.

With reference to the goals we outlined at the beginning of this section, we believe they are mostly achievable using the implementation techniques outlined. We stopped short of full Cronus integration in the design effort, by considering only databases that do not include other Cronus objects. More recently (see Section 3), we have begun work on providing database-like capabilities for Cronus objects as well. This work can serve as a basis for gradually introducing and experimenting with the above proposals for a more powerful and extensible data manipulation capability, active monitoring,

trading consistency and availability, and directory management.

RFFERENCES

See also Appendix A, which contains an annotated list of reports produced during the Cronus Distributed DBMS Project.

[Adiba 80] Adiba, M. E. and Lindsay, B. G., "Database Snapshots," *Proceedings Sixth International Conference on Very Large Data Bases*, 1980, 86-91.

[Bloch 87] Bloch, J., Daniels, D. and Spector, A., "A Weighted Voting Algorithm for Replicated Directories," *Journal of the ACM 34*:4, October 1987, 859-909.

[Bono 88] Bono, G., "Implementation Notes on the New Timeout Mechanism," Internal Report, BBN Laboratories Inc., June 1988.

[Chan 87] Chan, A Dayal, U. and Fox, S., "An Ada-Compatible Distributed Database Management System," *Proceedings of the IEEE 75*:5, May 1987, 674-694.

[Dayal 88] Dayal, U., "Active Database Management Systems," *Proceedings of the 3rd International Conference on Data and Knowledge Bases*, Jerusalem, June 1988.

[Floyd 88] Floyd, R., "Removing Timeouts From Cronus Object Invocations," DOS Note 114, BBN Laboratories Inc., March 1988.

[Gifford 79] Gifford, D., "Weighted Voting for Replicated Data," *Operating Systems Review 13*:5, December 1979, 150-163.

[Lindsay 84] Lindsay, B. G., Haas, L. M., Mohan, C., Wilms, P. F. and Yost, R. A., "Computation and Communication in R*: A Distributed Database Manager," *ACM Transactions on Computer Systems 2*:1, February 1984, 24-38.

[Orenstein 87] Orenstein, J. A. and Goldhirsch, D., "Extensibility in the PROBE Database System," *IEEE Bulletin on Database Engineering 10*:2, June 1987, 24-31.

[Parker 82] Parker, D. and Ramos, R., "A Distributed File System Architecture Supporting High Availability," *Proc. 6th Berkeley Workshop on Distributed Data Management and Computer Networks*, 1982, 161-183.

[Phadnis 88] Phadnis, N., "Multitype Query Processing in Cronus," in *The Cronus Distributed DBMS Project - Technical Papers Report*, Report No. 6974, BBN Systems and Technologies Corporation, December 1988.

[Rosenthal 86] Rosenthal, A. and Herman, P., "Understanding and Extending Transformation-Based Optimizers," *Database Engineering* 9:4, December 1986, 44-51.

[Sarin 85] Sarin, S. K., Blaustein, B. T. and Kaufman, C. W., "System Architecture for Partition-Tolerant Distributed Databases," *IEEE Transactions on Computers C-34*:12, December 1985, 1158-1163.

[Sarin 86] Sarin, S. K., "Robust Application Design in Highly Available Distributed Databases," *Proceedings Fifth Symposium on Reliability in Distributed Software and Database Systems*, January 1986, 87-94.

[Sarin 88a] Sarin, S., Floyd, R., Phadnis, N. and Vinter, S., "The Cronus Distributed DBMS Project - System/Subsystem Specification," Report No. 6858, BBN Systems and Technologies Corporation, December 1988.

[Sarin 88b] Sarin, S., "Integrated Access to Multiple Databases in Cronus," in *The Cronus Distributed DBMS Project - Technical Papers Report*, Report No. 6974, BBN Systems and Technologies Corporation, December 1988.

[Sarin 89] Sarin, S., Floyd, R. and Phadnis, N., "A Flexible Algorithm for Replicated Directory Management," *Proceedings of the Ninth International Conference on Distributed Computing Systems*, June 1989. (To appear).

[Stonebraker 86] Stonebraker, M. and Rowe, L. A., "The Design of POSTGRES," *Proceedings ACM SIGMOD Annual Conference*, May 1986.

[Vinter 88a] Vinter, S., Sarin, S., Floyd, R., Phadnis, N., Orenstein, J., Schroder, K., Turek, L. and Levine, R., "The Cronus Distributed DBMS Project - Functional Description," Report No. 6660, BBN Systems and Technologies Corporation, December 1988.

[Vinter 88b] Vinter, S., Schroder, K., Walker, W. and Levine, R., "The Cronus Distributed DBMS Project - Program Specification," Report No. 6854, BBN Systems and Technologies Corporation, December 1988.

[Vinter 89] Vinter, S., Phadnis, N. and Floyd, R., "Distributed Query Processing in Cronus," *Proceedings of the Ninth International Conference on Distributed Computing Systems*, June 1989. (To appear).

[Walker 88a] Walker, W. and Mackey, D., "The Cronus Library Catalog Search System," in *The Cronus Distributed DBMS Project - Technical Papers Report*, Report No. 6974, BBN Systems and Technologies Corporation, December 1988.

[Walker 88b] Walker, W., "CRSQL User Manual," in *The Cronus Distributed DBMS Project - Technical Papers Report*, Report No. 6974, BBN Systems and Technologies Corporation, December 1988.

[Walker 88c] Walker, W., "The Worker System," in *The Cronus Distributed DBMS Project - Technical Papers Report*, Report No. 6974, BBN Systems and Technologies Corporation, December 1988.

[Walker 88d] Walker, W., Vinter, S., Sarin, S., Floyd, R., Phadnis, D., Forsdick, H. and Mackey, R., "The Cronus Distributed DBMS Project - Technical Papers Report," Report No. 6974, BBN Systems and Technologies Corporation, December 1988.

[Walker 88e] Walker, W., "Cronus Associative Access Documentation," in *The Cronus Distributed DBMS Project - Technical Papers Report*, Report No. 6974, BBN Systems and Technologies Corporation, December 1988.

[Walker 88f] Walker, W., Vinter, S. and Forsdick, H., "The Cronus Calendar System," in *The Cronus Distributed DBMS Project - Technical Papers Report*, Report No. 6974, BBN Systems and Technologies Corporation, December 1988.

A. LIST OF CRONUS DISTRIBUTED DBMS PROJECT REPORTS

The following is the list of reports produced during this project:

- **Functional Description, BBN Report 6660**
  A document describing the functions of a distributed DBMS that supports highly available data.
- **System/Subsystem Specification, BBN Report 6858**
  A document describing the architecture and specification for the distributed DBMS.
- **Program Specification, BBN Report 6854**
  A document describing the integration of a commercial relational database management system within Cronus. This document describes two approaches toward integrating the DBMS: an interactive database service that makes the database query language visible to the client; and an embedded database service that defines an object-oriented interface to an application-specific service.
- **Interim Technical Report, BBN Report 6944**
  This document provides a technical status report for the project activities after the project was 75% complete.
- **Technical Papers Report, BBN Report 6974**
  A collection of papers generated during the project that do not fall within the scope of existing reports produced during this effort. These papers include:
  - Review of Candidate Database Management Systems for Integration into Cronus
  - The Cronus Calendar System
  - The Cronus Library Search System
  - The Worker System
  - The CRSQL User's Manual
  - Cronus Associative Access Documentation
  - Distributed Query Processing in Cronus (also to appear in the *Proceedings of the Ninth International Conference on Distributed Computing Systems*, June, 1989)
  - Multitype Query Processing in Cronus
  - Integrated Access to Multiple Databases in Cronus
  - A Flexible Algorithm for Replicated Directory Management (also to appear in the *Proceedings of the Ninth International Conference on Distributed Computing Systems*, June, 1989)

- **Final Technical Report, BBN Report 6973**
  This document describes the majors areas of work, the accomplishments of the effort, and directions for future design and development.

Several documents are available from BBN describing Cronus, including a Primer, a Tutorial, a User's Reference Manual, and a Programmer's Reference Manual. A general description of the Cronus architecture can be found in *The Architecture of the Cronus Distributed Operating System*, 6th International Conference on Distributed Computing Systems, Cambridge, MA, May, 1986, by Schantz, Thomas and Bono.

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control, Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C^3I systems. The areas of technical competence include communications, command and control, battle management information processing, surveillance sensors, intelligence data collection and handling, solid state sciences, electromagnetics, and propagation, and electronic reliability/maintainability and compatibility.