Object Properties: A Mechanism for Providing Runtime Services to Objects in a Distributed System

by

David Finkelstein
B.S. (Mathematical and Computational Sciences) Stanford University, 1986

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENT FOR THE DEGREE OF MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES
COMPUTER SCIENCE

We accept this thesis as conforming
to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA
October 1994

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

(Signature)

Department of COMPUTER SCIENCE

The University of British Columbia
Vancouver, Canada

Date OCTOBER 12, 1994

Abstract

Object-oriented systems are increasingly used as a means to develop distributed applications. Objects provide a natural unit of encapsulation for remote data, and the system can make remote invocations transparent to local users. Generally the underlying system provides a variety of services to objects in the system, such as persistence or concurrency control, which are used by developers of distributed applications. There are problems with existing mechanisms for providing such services, however: they may require the programmer to design a different subclass of each user class for every service available, limit the choices of services available, or inhibit performance by providing services to all objects even when not every sevice is needed. The work in this thesis attempts to solve these problems through a new mechanism for providing services to objects called *properties*. Properties allow services to be delivered transparently to objects on an as-requested basis. Additionally, a mechanism for describing inter-object relationships has been incorporated into the property scheme, allowing properties to be used to provide complex services such as atomic transactions. Properties were developed for the Raven distributed system and language developed in the Department of Computer Science at the University of British Columbia.

Table of Contents

List of Figures

Acknowledgment

First, I'd like to thank my family, especially my parents, for the support (both financial and otherwise) they've given me these past three years.

I'd like to thank Don Acton and Terry Coatta, my Partners in Crime (or Raven, as it were) for all the assistance they've provided me, from simply kicking around ideas to help in tracking down those elusive bugs. Without you two around, I had no other option but to graduate myself. I'd also like to thank my supervisors, Norm Hutchinson and Gerald Neufeld, not simply because you're supposed to do that sort of thing in your Acknowledgment, but for the guidance they've given me especially during those long meetings where I'd come up with strange scenarios involving object relationships. Thanks also to those who helped during the redesign of Raven, by contributing their ideas and providing me with feedback on mine: Stephan Mueller, Raymond Ng, and Jim Thornton. And a special thanks to Steve Loving, who helped encourage me in my dream to return to school.

CHAPTER 1

Introduction

1.1 The System Interface Problem

Computer operating systems are designed to present a virtual machine to the programmers and users of the computer. The operating system provides an interface through which applications can make requests to and receive services from the operating system. For example, operating systems usually provide a file service. By calling operating system routines, applications can find, read, and write files in the system. Many modern operating systems have a microkernel architecture, where the actual set of services provided by the operating system kernel itself are small. Instead, special servers provide the additional functionality of file service, networking, etc. Although the implementation is different, the basic model presented to the programmer is the same as that provided by more monolithic operating systems.

Unfortunately for developers of distributed systems, particularly systems designed for multi-threaded environments, the operating system interfaces can vary widely between different

operating systems. For example, the interface used by BSD Unix (or systems based on BSD Unix, such as SunOS) differs from that of Unix SVR4. Anyone who has attempted to port applications from one environment to the other can attest to the problems which can arise due to the differences in system calls. The interfaces to Mach and OS/2 are similarly incompatible with those of the other operating systems mentioned. These differences make the development of distributed applications more difficult. Attempts have been made to create a standardized interface for Unix systems. For example, many Unix systems support the POSIX application program interface [11], although most applications are still not written to this standard.

Another mechanism for providing a standardized interface is to use an object-oriented environment. Object-oriented systems allow the applications programmers to use a standardized interface to the object system. Since objects interfaces provide strict, well-structured accesses to the objects, they make a natural choice for providing the basis of a uniform system interface. It is necessary to port the object environment to each different platform, but once this has been accomplished, all user applications should be portable between platforms with a simple recompilation. Common object models have also been proposed and developed, primarily for the needs of object-oriented developers [14], [24].

1.2 Providing Services in Object-oriented Systems

Programming to a common object environment simplifies the development of applications in a distributed system. However, there remains the issue of exactly how user objects in the environment will receive system services. This is not a trivial question, as objects can require numerous services ranging from persistence to concurrency control to recoverability.

One mechanism is for the system to simply provide "system call" methods. In this scheme, object methods replace the system calls found in other operating systems such as Unix. This can be accomplished in one of two different ways:

(1) The system includes system objects which are responsible for providing services to user level objects. For example, objects which require concurrency control can invoke methods on a object which coordinates locking (e.g., a LockManager object):

```
LockManager.lockMe(self);
```

would ask the LockManager to lock the current object.

(2) The system provides classes from which objects can inherit behaviors which correspond to the services the object needs. For example, objects which require persistence can inherit methods from a Storage class:

```
self.storeMeToDisk();
```

would store the contents of the object to disk.

One system which uses this technique is Arjuna ([23], see Section 6.1.2). This system call mechanism has the advantage that objects can be tailored to receive only those services they need. However, it suffers from a number of problems. First, it requires the programmer to manage the use of the services. The programmer must invoke the appropriate methods to acquire and release locks, store the object to disk, etc. Although it's not necessarily a bad thing to make programmers responsible for managing the services they use, it does increase the likelihood of errors due to the programmer neglecting to release locks and from similar mistakes. This mechanism can also lead to the problem of *class explosion*: since the system provides multiple services, multiple versions of each user class might be required [1].

Another mechanism for providing services to objects in an object-oriented system is to provide a set of services automatically to all objects. In such a system, every object would be persistent, controlled against concurrent accesses, etc. This scheme has the advantage that the

programmer doesn't have to be concerned with managing the use of system services, they are provided transparently by the system. For example, the Guide system ([4], see Section 6.1.6) provides persistence to every object in the system. The major drawback of this approach is a lack of performance. Since every object receives services, the system must devote time and resources upon every object invocation.

Programmers and users ideally require a system which combines the best features of both mechanisms described above. Users of classes need to create instances that can receive any combination of system services, yet class designers shouldn't be required to include support for every possible service a user might desire in the class implementation, nor should there need to exist different versions of each class for every combination of services available. Users also desire maximum performance, requiring that invocations on the objects in the system be as fast as possible. For example, users should be able to create String objects which persist, which are controlled against concurrent accesses, or possibly both, yet objects which do not receive services should not suffer any undue performance penalty. Additionally, such a system should be complete enough to support distributed atomic transactions, a rigorous measure of the usability of a system.

1.3 Thesis Statement

An object-oriented operating system can associate each system service with a *property*, which, when assigned to an object, indicates the object should receive the corresponding service. The operating system can determine which services to provide to an object by an examination of the object's property set. The research presented in this thesis attempts to show that:

> *It is possible to identify and support a set of system services through object properties such that:*
>
> - *the semantics of the system services provided are independent from the others;*

- *services are provided on an instance-by-instance basis;*
- *services are provided automatically to an instance upon each invocation, without requiring any code in the class to use the service;*
- *the set of services provided to an instance can change dynamically throughout the lifetime of the object;*
- *the user can augment the set of services available;*
- *inter-object relationships can be described, allowing the system to provide complex services that span invocations on multiple objects;*
- *the services available, together with inter-object relationships, is sufficiently complete to support distributed atomic transactions yet does not limit the system to support only such transactions.*

1.4 Outline

A description of the object property scheme, including semantics for each system supported property, is found in Chapter 2. This chapter includes a general overview of Raven, the object-oriented operating system the property scheme was developed for, and discusses how the property scheme is used by Raven to provide services to objects. This chapter also discusses other aspects of the property scheme, such as the various inter-object relationships that were developed.

Details of the implementation of the property scheme are given in Chapter 3. This chapter describes how services are provided to objects via properties. Certain issues and problems encountered during the implementation are also discussed.

Chapter 4 describes the basic object storage service which was developed. It discusses the various issues confronted when designing the service, and provides details of the implementation.

A discussion of additional issues, and the future work which can be done to address them, can be found in Chapter 5. Many of the issues discussed in this chapter did not become apparent until later stages of the implementation or during testing.

Examples of other object-oriented systems, and the mechanisms they use for providing services to objects, are presented in Chapter 6.

Conclusions are presented in Chapter 7. A list of the data structures used by the runtime system to support properties is given in Appendix A. Class definitions for the various classes which were developed or modified in the course of implementing this thesis are given in Appendix B.

CHAPTER 2

Design of the Object Property Scheme

2.1 Introduction to the Raven System

The object property scheme was developed for the Raven system. Raven consists of a programming language and a runtime system which together form an environment for exploring issues surrounding distributed and parallel applications on both multiprocessor and distributed processor platforms. The development of the Raven system is an on-going project in the Department of Computer Science at the University of British Columbia. In addition to the work described in this thesis, Raven has been primarily used to investigate issues in parallel and concurrent programming [3] and to explore issues in configuration management of distributed systems [7].

The Raven language is an object-oriented language with a syntax similar to that of C. The language contains specialized constructs to directly support parallel execution of method invocations, some of which have repercussions for the various properties, as discussed in Section 3.7. The Raven runtime system provides support for objects written in the Raven language. Though

the syntax of the language is stable, significant enhancements to the runtime system are continually being made.

The complete Raven environment currently consists of a compiler, a class library, the runtime system, and a threads environment [16]. The Raven system is not very reified: very little of the runtime system, including the code that supports properties, is written in the Raven language itself. Instead, the bulk of the runtime system is written in C. Raven currently runs in its threads environment under Unix on Sun and MIPS workstations in a distributed environment. Raven is also running in a parallel environment under Mach 3.0 on a 20-processor Sequent machine. A microkernel running on multiprocessor MC88100 workstations is being developed to support the Raven system in a native environment [19]. A detailed overview of Raven, including a description of the language semantics, can be found in [1].

Each incarnation of the Raven environment is defined as a Raven World. Each Raven World has a corresponding UDP port number on the machine it is running through which it communicates with other Worlds. A Raven World is uniquely identified by its host identifier and port number. Different Raven Worlds, even those on the same local machine, are normally isolated from each other until the Worlds are informed of each other's existence.

2.2 Property Scheme Design Goals

The property scheme was designed with a number of goals in mind. The primary goal of the scheme is to provide a mechanism by which services can be provided transparently to objects on an instance-by-instance basis, in a flexible yet powerful way that does not limit the system or the programmer to a small set of possible services. The scheme was designed with each of the goals outlined below in mind.

2.2.1 Orthogonality of Property Semantics

When properties are designed so that their semantics are independent from each other, the properties are defined to be orthogonal to one another. The necessary strictness of the semantics has significant impact on the implementation of properties: properties can be designed, developed, and tested separately, and then added to the system without any concern that they might cause side effects. This is true whether the property being added is new, or simply a new implementation of an existing property.

Orthogonality is primarily useful because it makes the semantics easier to understand. The programmer doesn't have to be concerned with any changes to the semantics due to property interactions. When property semantics are independent, programmers can specify any combination of properties for an object with the knowledge that the object's behavior with respect to each property will always be correct. It is also an enabling feature; that is, when properties are orthogonal, it allows many other features to be included in the system, such as dynamic property assignment.

2.2.2 Properties Assignable Dynamically to Any Instance

To combat the problem of class explosion, the scheme was designed to allow properties to be assigned on a per-instance basis. In this way, sub-classes do not need to be created simply to gain the benefit of additional property behaviors. By allowing objects to receive services on a per-instance basis, performance can be improved: objects can receive only those services which they need, and so suffer no performance overhead related to services (such as persistence or concurrency control) which they may not require.

To further enhance the usefulness of the system, another goal of the design was to allow properties to be assigned dynamically to objects, even after the object had been created. Since properties are orthogonal, adding new properties to an object after instantiation can't create

unwanted side-effects. Dynamic property assignment can be useful especially in a client-server system. Consider an object server which creates objects for use by various clients. Different clients may require objects with different sets of properties. With dynamic property assignment, the property set desired doesn't need to be passed to the server. Also consider the situation where one application creates an object, a second application wants to use the object, but the second application needs the object to have properties which weren't required by the first application. For example, an object representing a block of text created by a text editor can be referenced by the body of a mail message. Although the text editor may not have created the text object with the persistent property, the mail handler can ensure that the object is persistent by dynamically assigning it that property.

2.2.3 Transparency of Use

Transparency implies that a programmer can implement classes without including special code to use properties. By placing all support for properties into the runtime system, the programmer is freed from the responsibility of asking for system services each time they are needed. This declarative model has advantages over the procedural model. The possibility of program errors with regards to receiving services from the system is greatly reduced, since the programmer does not have to be concerned with acquiring and releasing locks, writing object state to disk, etc., or determining when such events should occur. Additionally, even though the user of a class can assign any properties to instances of the class, the programmer doesn't have to be concerned with supporting all the properties in the method code (an unrealistic expectation, especially given that the properties available may change over time). This ties in well with orthogonality: a user of a class can create an instance of that class with any desired set of properties, without worrying about side effects between the properties desired or between those desired by the user and any additional properties specified by the class designer.

2.2.4 Support for Atomic Transactions

One primary goal of the system is for the property scheme to be powerful enough to support atomic transactions. It was a further goal that this could be accomplished without having to develop a special service devoted specifically to serializing invocations, but instead build the atomic transaction support using more general services.

2.2.5 Additional Support for Distributed Computing

Supporting atomic transactions allows the development of distributed databases and other robust applications. More simple applications, such as mail agents, do not require the same level of system support as a database. Other applications, such as name services, could benefit from system services which allow them to be highly available (e.g., some mechanism by which they could be replicated). As the property scheme was being designed, one of the goals was to provide sufficient services so that a wide variety of applications could be built. Additionally, the system needed to continue to support the experiments being conducted in the areas of concurrent programming and configuration management.

2.2.6 User Extensibility

To this point, properties have been described as a means of providing system-supported services to objects. The logical extension of this design is to extend properties so they can be used to provide objects with *user*-supported services. Such an extension allows users to develop their own services without the need to modify the runtime system to support them. This is extremely important, because the system properties could never provide all the services desired by users. It also allows potentially new system services to be designed and tested at the user level before being incorporated into the runtime system.

2.3 Selecting Services via Properties

Each of the goals enumerated above are important features for inclusion in Raven. To be truly useful, however, the properties should provide desirable behaviors when used in combination, and not simply in isolation. On the surface, this goal may appear to be incompatible with the goal of orthogonality. However, useful behaviors can be viewed as being composed of other, simpler behaviors. For example, an atomic invocation on an object requires recoverability, concurrency control, and persistence of the object state. Atomic transactions involve invocations on multiple objects and place additional requirements on the system, but it was a primary goal to design a property scheme that allowed atomic invocations to be performed by combining several properties together. If a new property had to be created for each desired behavior (e.g., if the system needed to support recoverability, concurrency control, persistence, and an "atomic invocation" property), then the number of properties could quickly grow to an unmanageable number (an undesirable property explosion, although growth would be linear and not exponential, as in class explosion), with a corresponding increase in the likelihood that the property semantics could not be designed in an orthogonal manner. For these reasons, considerable time was spent selecting the properties and developing their semantics.

The process of property selection began with an examination of a atomic transactions, to see if they could be supported by composing several services together. Fundamentally, atomic transactions can be broken down into three components:

- Serializability of invocations: If multiple transactions invoke methods on the same objects, there must be some serial ordering of invocations so that to each transaction it appears that it has exclusive access to the objects for the duration of the transaction.
- Recoverability: In the event of abnormal method termination (or possibly at the user's discretion) the system needs to be able to undo all the work that has been done so far.

- Persistence of data: Once a transaction commits, any changes that it performed must be permanent.

Attention was further focused to investigate the services which a single object would require for an atomic method invocation. Recoverability and persistence map directly as services required for an atomic invocation. The single-invocation analogue for serializability is concurrency control: since Raven is multi-threaded, objects must be protected against concurrent accesses. This provides a serial ordering of the invocations on the object. Consequently, the first three services chosen were concurrency control, recoverability, and persistence.

The semantics of persistence which are required in order to support atomic transactions must include strong guarantees about writing the object state to storage: once the method invocation finishes, the object state must be updated before control is returned to the caller. These semantics are necessarily very expensive to implement, since each invocation on a persistent object could result in I/O traffic. For many applications, it would be an acceptable compromise between persistence and performance to delay writing the data to disk for a number of seconds so the application can return from each invocation on a persistent object without blocking first. Machines and disks are generally reliable, so the chances of losing data by delaying the write a small number of seconds is extremely small. Many modern file systems, such as LFS [20], will buffer sequences of disk writes to memory in order to perform one larger disk write as opposed to many smaller writes. For objects which are being frequently modified, buffering writes in memory and only updating the version in non-volatile storage periodically can provide a desirable performance improvement. For these reasons, the system needs to support two types of persistence: one which provides a strong guarantee about when object state is written to disk, and one which provides greater performance by delaying updates.

In a distributed environment it is often advantageous to replicate data (such as name service information) on many different sites. This allows both an increase in performance (since clients

can access servers which are the most local) and availability (since the system is not dependent on the status of a single node). To support the development of highly available applications, a mechanism by which the system can make and maintain replicas of objects is required. Providing object replication was therefore chosen as another system service to support through properties.

To round out Raven's support for distributed computing, two other system services were chosen. First, since Raven's design allowed objects to be migrated between machines, some mechanism by which objects could be fixed to their current host (most likely their host of creation) was desired. Second, there are often objects which, once instantiated to a particular state, should not be modified. An example of such objects are those which comprise a mail message: once the message has been composed and sent, the objects should be considered to be "read only" and not modifiable. System support for creating immutable objects was therefore also desired.

In all, seven system services were chosen for support: concurrency control, recoverability, persistence with both strong and weak guarantees about storage updates, replication, immobility, and immutability. Security was not chosen as a service to provide for several reasons. First, Raven has no notion of a user, simply of the local Raven World; this precludes the use of access control lists or other mechanisms which provide access to objects on a per user basis. Second, the implementation of Raven allows only one capability structure for each object; this precludes the use of multiple capabilities, each with potentially different access rights. Third, little thought had been given to security during the initial development of Raven. Developing a notion of security for Raven is beyond the scope of this thesis, so all issues involving system support of security were left unresolved.

To correspond to the services provided, seven properties were chosen:

(1) Controlled

(2) Recoverable

(3) Durable

(4) Persistent

(5) Replicated

(6) Immobile

(7) Immutable

The Durable property corresponds to persistence with a strong guarantee that object state will be updated to storage, while the Persistent property corresponds to the weaker, more efficient notion of persistence. To receive a service from the system, an object needs only to be given the corresponding property.

By default, objects are created "plain", that is, without any properties. The Raven system handles plain objects in the following way:

- The object exists only in RAM. Therefore, the object does not survive between reboots of the system.
- No system control exists to prevent multiple threads from executing methods within the object simultaneously.
- The object can migrate from machine to machine, either at the discretion of the system or when the object is explicitly asked to move itself.
- Any changes made to the object's instance data are immediately available and are not recoverable.

For the vast majority of objects created and used in the system, these semantics should be all that are needed. Most objects will be created for short-term use and will not require any special

system properties. When an object is assigned one of the properties, the system handles the object differently; it is provided with the service which is associated with that property.

2.4 Property Behavior Semantics

One of the first questions encountered when deciding upon the semantics of property behavior was deciding at what level the properties would operate: should property behavior be limited to a single object invocation, or should it affect an entire invocation chain when the objects in the chain all have the same property? Each of these semantics has distinct advantages and limitations. Limiting the scope to a single invocation is a very convenient notion: properties can be described by how they effect one object, and can be viewed from the perspective of a single object. Limited scope is desirable in the general case for concurrency control: since locks are freed after each invocation, there is more concurrency and less potential for deadlock. However, it greatly complicates the issue of providing support for transactions, as a two-phase locking protocol cannot be used, and some other method must be used to enforce serializability. It would also require that the system be able to abort transactions at will: Consider a transaction *T1* which invokes a method on object *O*. After the invocation, a second transaction *T2* also invokes on object *O*. If *T1* aborts, then *T2* must also be aborted. Extending the scope to encompass all invocations is desirable in the general case for recoverability: if at some level the invocation can't continue, all the work so far should be aborted. If recovery simply restores the current object's state, programmers may become loathe to program using many small objects and instead encapsulate data into larger units. But extending the scope can also lead to programmer confusion, since it can quickly become unclear exactly how far up the calling chain each of the properties may be propagated. Furthermore, extending the scope does not eliminate the problem of providing semantics conducive to creating transactions.

The semantics decided upon came with the realization that there are two distinct concerns involved. The first involves objects and the system services they require, and the necessity to provide these services in a uniform manner. By viewing services from the point of view of the object, it becomes the natural choice to have limit the effects of properties to a single object invocation. The second concern is the understanding that objects are often placed in a relationship with each other, so that what happens to one object is predicated on what happens to the other. To provide for this, the property scheme requires a mechanism for describing inter-object relationships that allow the scope to be extended in the calling chain. This is described in Section 2.6.

For each of the properties, the semantics of behavior are described below. Since properties are orthogonal, the system will handle any object with any one of these properties in the same way, regardless of how many other properties the object has. Although it may appear on the surface that some of these properties are not orthogonal, that is not the case. Recoverable, Controlled, etc. provide only the semantics described, and do not attempt to provide more complex behaviors, such as serialized access or atomic invocations.

2.4.1 Controlled

An object given the Controlled property is protected against concurrent accesses by multiple threads which may modify its instance data. Multiple readers, but only a single writer are allowed access to the instance data. Threads which cannot be granted the access they require (e.g., they wish to write to the instance data while someone else is reading) are blocked until the request can be satisfied.

2.4.2 Recoverable

An object given the Recoverable property has the "all-or-nothing" property. Only when a method terminates normally are any changes made to the object's instance variables made permanent. If

the method terminates abnormally, the state of the instance variables is reset to the state they had just before the method started.

The Raven programming language includes a `restore` statement. If the current object is Recoverable, executing a `restore` will restore the object's instance variables to the state they were in before the method started. If the object is not Recoverable the `restore` statement is ignored. Execution continues with the next statement after the `restore`. Control is not returned to the caller.

2.4.3 Durable

An object given the Durable property has a copy placed in non-volatile storage (e.g., on disk). Any changes to the instance data of a Durable object are written to non-volatile storage before the method returns control to the caller. Furthermore, the system ensures that the entire object state is written atomically—only consistent, complete objects are written. It would be disastrous if only half the object state were written to storage, since upon restart the object image loaded in would be inconsistent. Return of control to the caller implies that the object state has been updated in storage. A Durable object's capability also survives system failure or restart. These semantics are necessary for the correct support of atomic transactions and database applications.

2.4.4 Persistent

The Persistent property is similar to the Durable property, except that no guarantee is made as to when the stored instance data will be updated. Although the in-RAM copy will be marked to be written to storage whenever a method modifies the object's instance data, Persistent objects are only written out periodically by the system to improve performance by reducing I/O traffic and increasing parallelism. It is not guaranteed that the current state of the object will be in storage at the time of a system failure, but the version in storage will be complete and consistent.

The semantics of Persistent are similar to those offered by the traditional Unix file systems, however Unix system semantics make no guarantee that the file will be written in a consistent state (i.e., some dirty pages may get written while others may not).

Although Durable and Persistent are very similar properties, the semantics of the two are different. Orthogonality still holds: if an object is both Durable and Persistent it properly obeys both semantics, since the semantics of Durable subsumes the semantics of Persistent. For most applications, such as mail agents, simple Persistence will be sufficient.

2.4.5 Replicated

An object given the Replicated property can be replicated on different machines. The decision to replicate is made by the runtime system in an effort to improve efficiency (for example, when an object is heavily accessed by processes on two different machines). Replicated objects have weak consistency: the system does not ensure that all copies of the object always have the same state. Replication is especially useful for improving performance and building highly available applications, such as name servers.

2.4.6 Immobile

An object given the Immobile property cannot be migrated between machines, and remains fixed on its current host. Immobility is desirable for objects which implement machine specific tasks, such as support for specialized devices or servers.

2.4.7 Immutable

The Immutable property prevents an object's instance data from being changed. If a thread attempts to invoke a write method (i.e., a method which can modify the instance data) on an Immutable object, a runtime error is generated.

Properties take effect only after the object has been created and its instance variables have been initialized. This permits an object which is created with the Immutable property to be brought into a usable state before the system prevents any accesses which could modify the instance data.

2.4.8 User-Defined Properties

The seven system properties described above are each implemented directly by the system. Although it is not possible for a programmer to modify the implementation of any of these properties, programmers can define semantics for and create implementations of their own properties. The user-defined properties are assignable in exactly the same ways as are system properties.

User-defined properties, by their nature, can only affect data in user space; that is, their effects are identical to invoking methods on objects. If the programmer defines several properties, it becomes the programmer's responsibility to maintain the orthogonality of the properties.

2.5 Assigning Properties to Instances

Since objects in the Raven system can have any combination of properties, a programmer needs to specify which properties an object will have. The Raven language provides keywords for specifying the different properties:

- Controlled property: `controlled`
- Recoverable property: `recoverable`
- Durable property: `durable`
- Persistent property: `persistent`
- Replicated property: `replicated`
- Immobile property: `immobile`
- Immutable property: `immutable`

- Test property: `test_prop`
- User properties: `u_prop_1`, `u_prop_2`, `u_prop_3`, `u_prop_4`

Object property specification can be done in two ways:

(1) The class designer can specify that all instances of the class will have certain properties. The class designer does not need to write any code to support the properties, since they are all supported by the system.

As an example, consider the code fragment from a class definition shown in Figure 1. All instances of the class HappyObjects would be given the Controlled and Recoverable properties.

```
class HappyObjects controlled recoverable
{
    someInt : Int;
    behav doSomething();
    ...
}
```

FIGURE 1. Assigning properties in the class definition.

(2) At object creation time, the programmer can specify a list of properties for the object. Objects in Raven are created using one of two different methods. The `new` method returns an instance of the class, which will have only those properties specified by the class designer. The second creation method is `pnew`, which takes a list of additional properties to assign to the instance as one of its arguments.

Consider the code fragment shown in Figure 2, which shows an assignment to the instance variable someObject. Since someObject is of class HappyObjects, it will have the Controlled and Recoverable properties, as well as the Persistent property and a user-defined property.

```
behavior someBehavior
{
    ...
    someObject = HappyObjects.pnew(persistent & u_prop_1,
                                   arg1, ...);
    ...
}
```

FIGURE 2. Assigning properties by using pnew.

Although the runtime system supports dynamic property removal (since it must be able to remove properties from an object when the object no longer inherits them) there is no syntax in the programming language to allow programmers to dynamically remove properties from objects.

2.6 Inter-Object Relationships

As described in Section 2.4, property semantics were developed by viewing properties as they applied to a single level invocation on an object. But objects do not exist simply in isolation, and therefore some mechanism by which the relationships between objects can be described is necessary in order to provide support for more complex behaviors like atomic transactions. In addition, to achieve the goal of dynamic property assignment, some mechanism must also exist by

which objects can be placed in some relationship that provides the dynamic assignment of properties from one object to the other. Towards these ends the Raven language was modified to provide three methods of describing relationships between objects: *Dependent* references, *Inherits* references, and *Part-Of* references. When the designer of a class wishes to describe an inter-object relationship, a class instance variable can be marked as either `dependent`, `inherits`, or `partof`. Consider the class definition shown in Figure 3. In this example, `firstInstanceVar` is a Dependent reference, `secondInstanceVar` is an Inherits reference, and `thirdInstanceVar` is a Part-Of reference.

```
class ExampleClass
{
    aFirstInstance : cap dependent;
    aSecondInstance : cap inherits;
    aThirdInstance : cap partof;
    ...
}
```

FIGURE 3. Class definition with instance variables showing inter-object relationships.

2.6.1 Dependent References

Dependent references are the tool by which the scope of a property is extended beyond a single invocation to encompass a calling chain. A Dependent reference is defined as follows. When an object *P* has a reference to another object *C* which is marked `dependent` (they can be thought

of as Parent and Child objects), *C* is said to be Dependent on *P*. The dependency refers to *property dependency*: the properties of the child are dependent on those of the parent. When the parent invokes a method on the child, state information associated with the properties is passed upwards to the parent when the method returns. Any actions that would normally be taken before the return from the child become dependent upon the parent.

Dependency has meaning only for properties; as such, it does not affect plain objects. In the current design of Raven, the semantics for Dependent references affect only the Controlled, Recoverable, and Durable properties. It makes little sense to speak of "Dependent Immutable" or "Dependent Immobile", although some semantics for these situations could be contrived; also, since Replicated only provides weak consistency and Persistent makes no guarantees as to when instance data will be written to storage, no semantics for Dependent Replicated or Dependent Persistent have currently been devised.

As examples of how Dependency works, consider the following situations, where the object *P* has a Dependent reference to an object *C*, and *P* invokes a method on *C*:

- If *P* and *C* are both Controlled, then the access privileges (read or write privileges) associated with the invocation on *C* are retained by the thread which made the invocation and are passed back to *P*. Other threads are still blocked from accessing *C*. Since the thread now in *P* still holds access privileges, it can make subsequent invocations on *C* with impunity (assuming that the access rights held are sufficient for the invocations—if the initial invocation only required read privileges, the first future invocation that requires write privileges will block if other readers are present). The thread relinquishes control of the object *C* when its invocation on *P* returns, at which time the thread's control of *P* is also relinquished.
- If *P* and *C* are both Recoverable, then any changes made to *C* are not committed until the changes made to *P* are committed (i.e., the invocation on *P* returns normally). If *P* returns abnormally, or an `restore` statement is executed, then

both *P* and *C* are restored to the states they were in before the invocation on *P* began.

- If *P* and *C* are both Durable, then any changes made to *C* are not written to disk until the changes made to *P* are written. The invocation on *P* will not return until both *P* and *C* have been written. Since the Durable property assures that objects are written in their entirety, it is guaranteed that both *P* and *C* will be written in an atomic fashion.

If *P* is a dependent child of a third object *G*, then the state information for the properties of *C* and *P* are passed to *G*. State information about a property is kept by the thread, and passed back from a child to a parent until the top level of the Dependent chain is reached. Raven places no limit on the number of Dependent references that may exist to an object.

2.6.2 Property Inheritance

Section 2.5 described how properties are assigned at the time of object creation. An additional way in which an object can receive properties is dynamically at runtime through property inheritance. When an object is assigned to an instance variable which has been marked `inherits`, the object is then assigned all the properties of the object holding the reference. Since the object being assigned can also have instance variables which are marked as `inherits`, the runtime system must compute the transitive closure of all the references which are marked `inherits` and update the properties of all the objects so referenced. The complete set of properties of an object are those assigned at creation time plus those it inherits from another object. In the current design of Raven, an object can inherit properties from at most one other object, and there is no way to "mask out" certain properties, i.e., to specify that an object should not inherit a property dynamically. The design also does not include a mechanism by which objects can be prevented from acquiring properties dynamically, nor a mechanism by which the class designer can specify that instances of the class should never be given certain properties.

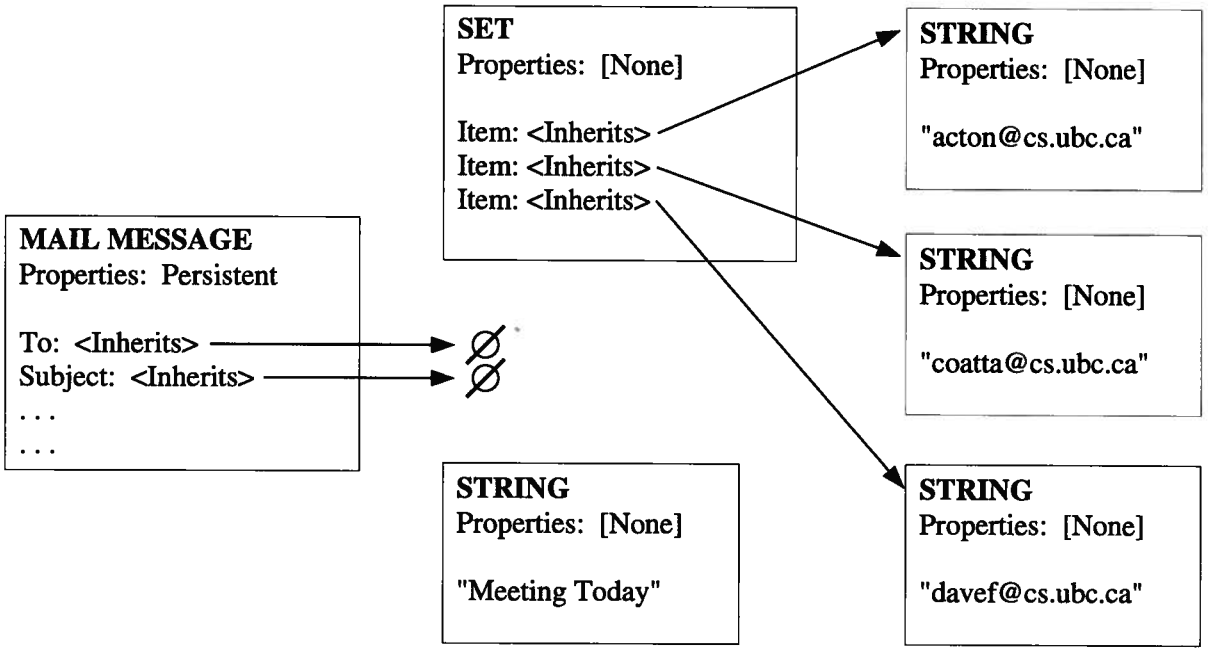

FIGURE 4. Some of the objects comprising a mail message, just prior to being assigned to the MAIL MESSAGE

As an example of the use of property inheritance, consider a mail message. All the objects which make up the mail message (e.g. the `To:` and `Subject:` fields) need to be Persistent. However, it is sufficient to simply create the main mail message object as Persistent, and have each of the sub-objects inherit Persistence from the main object. The advantage of such a scheme is obvious when you consider that objects that comprise the mail message, such as Sets or Strings, may have been created by other applications (such as an editor) and may not have been created with the Persistent property. (See Figure 4 and Figure 5.)

The ability to dynamically assign properties to an object is important in client-server systems where servers create objects for use by different clients. With dynamically assignable properties, clients can be assured that the objects they use will have those properties they need. The behavior of the object can be specified by the user of the object, and the creator need not know in advance which properties to assign.

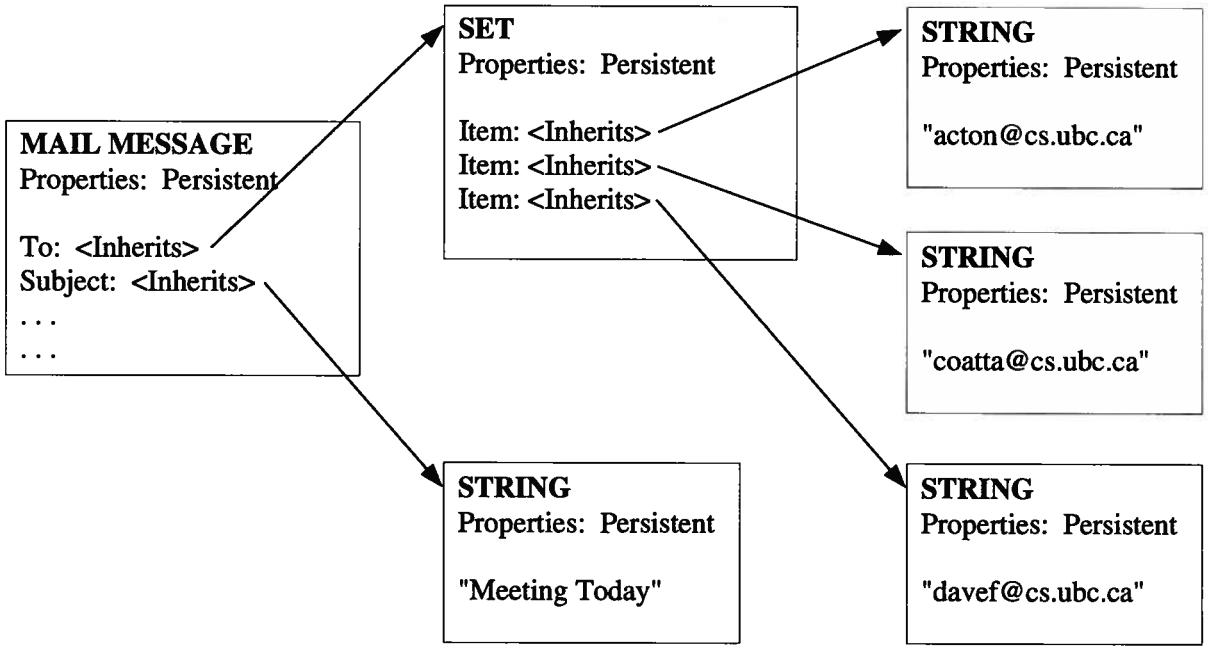

FIGURE 5. After assignment, objects have new properties and behave accordingly.

Property inheritance should in no way be confused with the traditional notion of inheritance which describes the inheritance of methods and behaviors by one class from another. Property inheritance is a relationship between two objects, indicating that additional system services should be bequeathed to an object. Objects can inherit properties from other objects of any class.

2.6.3 Part-Of References

Dependency describes a relation between objects which affects how the system handles objects with properties. In object-oriented systems, objects are often bound closely together. Consider again a mail message object. The mail message is composed of many other objects: a `From:` field, a `To:` field, etc., each of which are themselves often composed of many other objects—e.g., the `To:` field could be a Set or a List or just a simple String object. In such circumstances, the sub-objects make little sense in an isolated context. They are intrinsically part of the mail message object. This relationship goes beyond simple Dependency; the objects are tightly coupled together

by the relationship the programmer has created between them. To describe such a relationship between objects, Raven allows class instance variables to be marked `partof`.

By specifying a reference as `partof`, the reference is considered to be marked as both `inherits` and `dependent`. For the programmer, there is no semantic difference between specifying a reference as `partof` and specifying it as both `inherits` and `dependent`.

The system interprets Part-Of to imply a tight coupling between the objects. As such, the Raven runtime system can make special use of Part-Of relationships. Consider a chain of objects, $A \rightarrow B \rightarrow C$, where C is Part-Of B and B is Part-Of A. Together, A, B, and C can be viewed as a single cluster by the Raven system. Since the objects in a cluster are tightly coupled, object clusters can be used by the Raven system in a variety of ways:

- Object clusters can be written to disk as a single unit, even if they are not contiguous in memory. Similarly, when the root object of an object cluster is loaded in from disk, the entire cluster is loaded, since it is probable that many of the objects in the cluster will soon be needed.
- Object clusters can be migrated together between machines. Simply migrating the root object of a cluster would be inefficient, since any invocations by the cluster root object on other objects in the cluster will either have to be remote (i.e., across machine boundaries), or cause additional object migrations (with their associated overhead).

Clusters are an efficiency mechanism for the runtime system and do not alter the semantics for the programmer. Because objects can inherit properties from at most one other object, an object can be Part-Of at most one other object. The Raven system does not limit the number of references that can exist to an object that is Part-Of another.

As a possible use of Part-Of, consider again the mail message objects of Figure 4, only this time with instance variables marked `partof` instead of `inherits`. Performance would suffer

greatly if the system only loaded in the root object from disk when the mail message was accessed, or only migrated the root object to another machine; in a very short time, the other pieces of the object will need to be accessed. By logically clustering the objects together the overall performance of the system can be improved.

It should be emphasized that Part-Of is used by the runtime to provides performance optimizations only, and is otherwise no different than specifying `inherits` and `dependent`.

2.7 Combining Properties

Since properties in the Raven system are orthogonal, they can be easily combined. No side effects result from adding properties to objects; instead, the system simply provides the new functionality of the added property. Properties can be combined to produce the exact behavior required, while invocations on the object do not incur any overhead for system services not used.

Although many objects will have only a single property (e.g. Controlled or Persistent), more complex applications and objects require several properties. One natural pairing is Immutable and Replicated. Since an Immutable object cannot be modified, it can be replicated on many machines (or even in several places on the same machine) without any concern for maintaining consistency between copies.

Different applications will require different pairings of properties. Consider a name service. The name service must support concurrent lookups by multiple users, provide a facility for adding and removing names, while maintaining the integrity of the service across system reboots. The objects which comprise the name server need to be both Controlled and Persistent, otherwise the name server will not function properly.

2.7.1 Supporting Atomic Transactions

More complex systems require more properties. By combining Raven properties and placing objects in a Dependent relationship, it is possible to create database systems which support atomic transactions. Objects in the database need to be Controlled, Recoverable, and Durable. By putting the objects in a Dependent relationship, the invoking thread has exclusive access to the objects until the transaction returns, as the thread will retain all the concurrency control locks for the objects touched. New locks can be acquired, but old locks will not be released until the top level invocation returns. This behavior is analogous to a two-phase locking protocol, and therefore provides the serializability required for atomic transactions. Since the objects are Recoverable, any abnormal termination or if a `restore` statement is executed will restore the state of all the objects touched to the values they had before the transaction began. Finally, the new state of all the objects will be written to storage at the same time, in an atomic fashion, prior to the return from the top level invocation. As such, in the event of a system failure, either all the objects are updated in storage, or none of them are.

CHAPTER 3

Implementation of the Object Property Scheme

In order to test the ideas developed in the previous chapter, core features of the property scheme were implemented and integrated into the Raven runtime system.

3.1 Data Structures for Property Support

An object in Raven is referenced using a capability pointer to the object. These pointers are normal 32-bit integer pointers to memory. Capability pointers point to a block of memory which contains a capability structure. These structures contain the data, or pointers to the data, used by the runtime system for the management of the object, as well as a pointer to a block of memory which contains the object's actual instance data. The capability structure is shown in Figure 6.

An object's property set is stored as an integer value inside the capability structure. This 32-bit value is divided into two 16-bit words. The first word is used to store the set of properties the object was assigned at creation, while the second stores the currently inherited properties. This

```
struct capability
{
    funcptr         invoke;
    cap             id;
    cap             is_a;
    cap             parent;
    cap             inh_root;
    cap             storage_manager;
    method_type     method_type_to_use;
    struct gid      *gid;
    voidp           rw_lock
    voidp           cluster_lock;
    properties      object_properties;
    u_char          *data;
};
```

invoke:	Pointer to the invocation function to use
id:	Pointer to this structure, for sanity checking
is_a:	Capability of class object which we are an instance of
parent:	Capability of object we inherit properties from
inh_root:	Root object of inheritance tree
storage_manager:	Capability of object which manages our storage
method_type_to_use:	Indicates if a remote invocation is needed
gid:	Pointer to object global identifier
rw_lock:	Pointer to object concurrency control lock
cluster_lock:	Pointer to object cluster lock
object_properties:	Object property set
data:	Pointer to object instance data

FIGURE 6. Object capability structure.

design limits the number of properties that the system can support to 16—each bit position indicates whether the object has a particular property or not. In addition to the seven system properties and the four user properties, Raven provides a "test" property that was used extensively during the testing of property inheritance and dependent invocations. The test property can also be used as the basis for testing new system properties before modifying the compiler and the runtime system

to support the new property name. In total 12 bits of each word are accounted for, allowing up to four more to be added before these fields would need to be widened.

As described in Section 2.5, a programmer can specify new properties for an instance by providing a list of properties as an argument to the `pnew` method. It is a natural convention to list these properties using an "&" operator, as in `pnew(prop & prop & prop...)`: when a programmer wants an object to have "concurrency control and recoverability" it can simply be written as "`controlled & recoverable`". To facilitate this, each of the Raven properties is represented as a bitmask of all 1s, with a 0 in the position corresponding to that property. The objects property mask is then created by performing a bitwise-and of the properties the object was given. If an object is created with a particular property, it will have a 0 in the corresponding position of the first word of its property mask; if it inherits a property, it will have a 0 in the corresponding position of the second word of its property mask. By storing properties as bitmasks, the runtime system can easily detect when an object has a particular property. The bitmasks are given in Appendix A.2. Simple macros have been written to test and set the appropriate bit positions for each of the properties.

The other pieces of the capability structure used by the runtime system to support properties are:

- The `parent` capability: Points to the capability structure of the object from which properties are inherited. If no properties are currently being inherited, this value points to the `nil` object.
- The `inh_root` capability: Points to the root object of the property inheritance tree. Note that not all inherited properties may be from the root object; some may be from an intermediate object.
- The `storage_manager` capability: Points to the object which manages the storage of this object to disk (see Chapter 4).

- The `gid` pointer: Points to a data structure containing the object's globally unique identifier (see Chapter 4).
- The `rw_lock` pointer: Points to a lock data structure used for concurrency control, or NULL if the object is not Controlled.
- The `cluster_lock` pointer: Points to a lock data structure used for Part-Of cluster locking (see Section 3.5).

3.2 Providing System Services Through Properties

The purpose of properties is to allow system services to be provided transparently to objects. Services are provided whenever a method is invoked on an object. To accomplish this, the runtime system uses a series of pre- and post-invocation functions, with one set of functions for each of the properties. Immediately prior to the method invocation, the property set of the object is examined. For each property the object has, the appropriate pre-invocation function (termed a pre-handler) is executed. When the necessary pre-handlers have been executed, the method code itself is executed. After the method returns, prior to returning control to the calling object, the appropriate post-handlers are executed.

The order in which the pre- and post-handlers are executed is very important to ensure correctness. For example, it is necessary that locks not be released until after the Durable and Recoverable post-handlers have executed, since these functions may need to read or write the instance data of the object; it would be disastrous if another thread began modifying the object's instance data while the Durable post-handler was attempting to write the data to disk. Correctness places several restraints on the ordering that can be used, the primary one being that concurrency control locks must be acquired before any other work is done and released only after all other work is finished. Correctness also requires that the Recovery post-handler execute first, since it may restore the object state. They system must not propagate an incorrect state to replicas of the object, or store an incorrect state to disk.

The current ordering of the pre-handlers is as follows:

(1) Controlled

(2) Durable

(3) Persistent

(4) Replicated

(5) Test property

(6) Recoverable

(7) User properties (u_prop_1 through u_prop_4)

The post-handler order is the exact inverse of the pre-handler order.

There are no pre- and post-handlers for the Immutable and Immobile properties. To support the Immutable property, the runtime system checks the method type just prior to executing the Controlled pre-handler. If the method is a write method (i.e., if it modifies the object's instance data), then a runtime error is generated. The determination of the method type is done by the Raven compiler through an analysis of the statements in the method. Although the compiler can be fooled and generate an incorrect method type, and some invocations of write methods may not in fact modify the instance data, this implementation is extremely simple, and provides a reasonable implementation. The Raven runtime system does not currently implement object migration, and as such, the Immobile property is not used. However, when migration is implemented, immobility will still not need a pre- or post-handler, as it will simply be checked for just prior to object migration.

Since the property set of an object can change dynamically, it's possible for one thread to be reading the property set as it begins an invocation while a second thread is modifying the property set as the result of its own invocation. This problem is addressed by several features of the runtime. First, the property set is stored as an integer value. It can be assumed that the hardware can

read and write integer values to memory atomically; as such, a thread reading the property set will not see a partially updated property set that is being written by another thread. Second, a thread must have the object's concurrency control lock in order to modify the property set; if no lock exists, then that thread can be assumed to have exclusive access to the object. Since the concurrency control lock is acquired before any work is done (even work required by the other properties), it can be assumed that the property set of an object will remain stable for a thread after it is granted the lock (unless the thread modifies the property set itself). If the property set was changed while threads are waiting to be granted the object's concurrency control lock, they are restarted, object immutability is rechecked, and the threads once again begin the process of executing the pre-handlers.

One alternative to using a careful ordering of the pre- and post-handlers is to have the handlers executed atomically—that is, require that no other threads be allowed to execute while one thread is inside the set of pre- or post-handlers. Using a careful ordering provides better performance, but it also precludes the use of user-defined properties for providing alternate forms of some of the system services, particularly concurrency control, since no ordering could allow both system-supported concurrency control and user-supported concurrency control to be the first pre-handler executed. However, if the handlers were executed atomically, some mechanism would still be required to deal with the problem of dynamically changing property sets, and potentially undoing work done in a pre-handler for a property which the object will not have when the method is actually executed.

An examination of the pre- and post-handlers helps demonstrate the orthogonality of the properties. The handler code for each of the implemented properties makes no references to any other property, nor does the code access data structures used by the pre- and post-handlers of the other properties. The exception to this rule is the handlers for the Durable and Persistent properties, which perform almost identical tasks. Although separate implementations could have been

made, it would have required an almost complete duplication of the existing data structures and code. It was therefore decided to combine the implementations and allow them to share data structures and supporting objects (see Chapter 4).

3.3 Dependent Invocations

Whenever a method is invoked on an object which is referenced via an instance variable which has been marked `dependent`, the runtime system must keep track of state information associated with the properties. This is accomplished by maintaining this state information in the thread. Currently, the runtime system uses the thread to thread keep track of information for the Controlled, Recoverable, Persistent, and Durable properties; these properties therefore support Dependent invocations. Each thread has associated with it a corresponding thread object (an instance of the Raven `Thread` class). Thread objects include among their instance variables the following:

- `session_chain`: An integer value which is used by the runtime system as a pointer to a list of lock structures currently held by the thread.
- `lock_depth`: The integer value of the current depth in the invocation chain, starting at the top-most Controlled object and counting only Controlled objects in the chain. `lock_depth` is basically the count of the number of locks held by the thread.
- `shadows`: An integer value which is used by the runtime system as a pointer to a list of object shadow copies, i.e., copies of object instance data. These shadow copies are created for Recoverable objects and are used to restore object state.
- `callDepth`: The integer value of the current depth in the invocation chain, starting at the top-most Recoverable object and counting only Recoverable objects in the chain.

- `function_chain`: An integer value which is used by the runtime system as a pointer to a list of functions that need to be executed at some later time. The use of `function_chain` is discussed in Section 3.4.1 and Section 3.5.3.
- `storage_chain`: An integer value which is used by the runtime system as a pointer to a list of objects which need to be written to storage. `storage_chain` is discussed in Section 4.4.

Each Dependent invocation has associated with it a corresponding identifier. A chain of Dependent invocations all use the same identifier, which is set to the `id` (integer value of the capability pointer) of the root object of the Dependent chain.

When an invocation is made in Raven, a parameter structure is passed to the invoke routine. This parameter structure contains data used in the course of the invocation, such as the method name, and the actual parameters used by the method. These parameter structures are pushed onto and popped off of the C-level stack with each invocation. The runtime system can examine the current parameter structure for its current invocation, or trace upwards through the parameter structures for previous invocations. The parameter structure contains a boolean flag, `dependentInvoke`, that indicates if the current invoke is Dependent or not. The Raven compiler checks each invocation to see if the invokee is Dependent upon the invoker; if so, the compiler emits code to set this flag to `TRUE` (a predefined boolean value used by the runtime). This allows the runtime system to know when a Dependent invoke is occurring. Details about invocations in Raven, including the use of parameter structures, can be found in [1].

The parameter structure contains another flag, `isDependentRoot`, which is normally set to `FALSE`, as well as an integer field for the Dependent invocation identifier, `dependentID`. When the runtime finds the `dependentInvoke` flag set, it examines the parameter structure for the previous invocation. If this structure has a non-zero value for its `dependentID`, this value is used as the Dependent invoke identifier of the current invocation. If this value is zero, then the

invoker must be the root object of the Dependent invocation. The `isDependentRoot` flag is set to `TRUE` in the invoker's parameter structure, and the local `dependentID` is set to the `id` of the invoker.

Normally, post-handlers are executed only for properties which an object has. However, it is possible that the root object of a Dependent invocation chain does not have the Controlled, Recoverable, or Durable properties, yet objects with these properties were invoked somewhere in the chain. The post-handlers for these properties must therefore be executed when the invocation on the root object has completed. If the `isDependentRoot` flag is set, these post-handlers are executed. The post-handlers check the value of this flag, and perform the necessary work if Dependent invocations were made on objects with the corresponding property (see Section 3.3.1, Section 3.3.2, and Section 4.4).

A Dependent invocation chain only includes objects which are Dependent upon their callers. If an invocation is made on a non-Dependent object, then any further invocations made from that object will be part of a new, separate Dependent chain. For example, consider a chain of invocations, $A \rightarrow B \rightarrow C \rightarrow D \rightarrow E$, where E is Dependent on D and B is Dependent on A. Both the invocation on A and on D are considered to be roots of Dependent invocations, and will have their `isDependentRoot` flags set. Properties which support Dependency need to keep track of the `id` of the current Dependent chain, to ensure that the post-handler does not do work prematurely: using the above example, when the invocation on D terminates the Durable post-handler needs to write D and E to storage, but not A and B.

3.3.1 Dependent Invocations on Controlled Objects

Each invocation on a Controlled object has an associated lock depth, which is basically a count of the number of locks the thread currently holds. An initial invocation on a Controlled object has a lock depth of one; an invocation it makes on a Controlled object would have a lock depth of two,

etc. The lock depth is unaffected by invocations made on objects which do not have locks: if *A* invokes on *B* which invokes on *C*, and only *A* and *C* are Controlled, the lock depth of the invocation on *C* will be two. The locks held by the thread are kept in a list referenced by the `session_chain` instance variable of the thread object. To support Dependent invocations, the lock information data structure described in [1] was augmented to include a field for the value of the current Dependent invocation identifier; this field is also called `dependentID`. The value of this field is set by the Controlled pre-handler to the current value of `dependentID` found in the parameter structure.

When a method terminates, the post-handler checks the `dependentInvoke` flag. If it is set, then the handler simply returns without releasing any locks; if not, then locks are released up to the current lock depth. If the `isDependentRoot` flag is set, then the post-handler must release all locks that were acquired during the Dependent invocations. If the current object is Controlled, all locks with a lock depth greater than the current depth are released, since all the locks acquired after the current lock will have a lock depth greater than the current lock depth. If however the current object is not Controlled, then the `session_chain` is traversed starting from the most recently granted lock. If the `dependentID` of the lock information structure is equal to the `id` of the current object, the lock is released, and the next lock in the chain is examined. This process continues until no locks remain in the chain, or a lock information structure has a `dependentID` not equal to the `id` of the current object.

3.3.2 Dependent Invocations on Recoverable Objects

When Recoverable objects are part of a Dependent calling chain, the shadow copies of the objects' instance data must be retained by the thread until the top level invocation in the Dependent chain terminates. The thread also keeps track of the current depth in the invocation chain using the instance variable `callDepth`.

Each time an invocation is made on a Recoverable object within a Dependent calling chain, a shadow copy of the object is made and placed on the `shadows` list kept by the thread. The current value of the `callDepth` is then set for this invocation, and stored with the shadow. When the top level invocation terminates, the shadow copies are simply discarded, as they are no longer needed. However, if a `restore` statement is encountered, then the list of shadows is traversed and all objects whose shadows have a higher call depth than that of the current object have their instance data restored to the value of the shadow, and these shadows are removed from the shadow list.

3.3.3 Remote Dependent Invocations

Each time a remote invocation is made in Raven, a worker thread is created on the remote machine to execute the method code [1]. Normally, this thread is destroyed after the method finishes executing. However, since property state information is kept in the thread, remote Dependent invocations must be treated in a special manner. Additional problems arise due to the nature of the implementation of properties: since most of the data structures used are regular C structures and pointers, references to the state information cannot be easily passed back to the original machine. Propagating the state data back can be expensive, and troublesome to maintain, as changes are made to the data structures or support for Dependency is added to additional properties. To address these concerns the following solution to the problem of remote Dependent invocations was adopted.

Every remote invocation now includes an additional integer parameter in which the ID of the current Dependent calling chain is passed (or zero if the invocation is not Dependent). A potential problem can occur, however, since Dependent IDs are just the location in memory of the root object of the Dependent chain, and there may be an object at that location on the remote machine which gets invoked as part of the remote invocation. Therefore, the actual ID passed is (`<real Dependent ID> | 1`), as no object could ever reside at an odd memory address.

When the remote thread finishes execution of a Dependent invocation, in addition to any return value it also passes back the globally unique ID (`GID`, see 4.1.1) corresponding to the `Thread` object of the remote thread. The remote thread then suspends itself, so the `Thread` object and its associated data structures are not destroyed.

The `gid` of the remote `Thread` object is stored with the state information for each property in the local thread. When the invocation on the root object of the Dependent invocation chain finishes, the property post-handlers are executed. As the post-handlers traverse their state data, if a reference to a remote `Thread` object is encountered then a remote invocation is made on that object (see Appendix B.1). The remote worker which is created does not awaken the original, suspended worker; instead, it simply uses the worker's instance data to perform the appropriate work necessary for the specific post-handler. This is possible because all remote worker threads of an original thread have the same session identifier as the original thread, and this session identifier is supposed to be unique for all threads in the system. Once all the post-handlers have executed, the original worker is awakened and is then destroyed.

Currently, the system stores the remote `Thread` object `gid` in the state information of every property, even if no objects with that property were invoked upon by the remote thread. Therefore, remote Dependent invocations will require one additional remote invocation for every property which supports Dependency.

3.4 Property Inheritance

In the general case, supporting property inheritance is fairly straightforward. The Raven compiler detects any assignments which are made to a class instance variable which is marked with the `inherits` keyword, and emits code to call a special runtime routine, `UpdateInherits()`, which does the work of "disinheriting" properties from the old object which had been referenced

by the instance variable and "bequeathing" properties to the new object assigned to the instance variable.

The basic steps taken in disinheriting properties from and bequeathing properties to a target object (i.e., an object which has been de-assigned from or assigned to an instance variable which provides property inheritance) are the same. When bequeathing properties, two additional steps must be taken: first, the system must ensure that the target object is in the same Raven World as the bequeathing object; and second, the system must check to see that a property inheritance cycle is not being created, by checking to see if the target object is the root object from which the bequeathing object inherits properties.

The system first computes the transitive closure of all objects which inherit properties from the target object. These objects are then locked to prevent any concurrent accesses to them while their properties are being updated. Next, the property mask maintained in each object's capability structure (see Figure 6) is recomputed, and any work associated with the gaining or losing of properties (e.g., creating or destroying locks used by the Controlled property) is performed. The value of `inh_root` stored in each object's capability structure is also updated accordingly. This pointer points to the root object of the property inheritance tree; an object which no longer inherits any properties has its `inh_root` set to point to itself. The value of the `parent` pointer for the target object must also be updated; this is either the object from which properties are directly inherited, or `nil` if no properties are inherited. Finally, the locks on the objects are released.

Currently, the system does not do any performance optimizations, such as checking to see if disinheriting or bequeathing properties actually affects the property sets of all the objects in the transitive closure, and then omitting the unaffected objects from the set of objects that need to have their property masks updated.

3.4.1 Recovering Property Inheritance Assignments

Although providing property inheritance is straightforward in the general case, it becomes much more complicated when the assignment which provided property inheritance can be recovered. Consider the example shown in Figure 7, where `anInstance` provides property inheritance.

```
behavior doSomething()
{
    ...
    if (anInstance == oldObject)
        anInstance = newObject;
    ...
    restore;
    ...
}
```

FIGURE 7. An Example of Recovering a Property Inheritance Assignment.

The original object (`oldObject`) which had been referenced by `anInstance` and all objects which inherit properties from `oldObject` will have their property masks changed, and `newObject` and all objects which inherit from `newObject` will have their property masks changed. Furthermore, these objects must behave as if they have (or don't have) the appropriate properties; if `newObject` was uncontrolled before, but has the Controlled property now, it must be given a lock. However, once the restore statement is executed, all the work which was done to disinherit `oldObject` and bequeath properties to `newObject` must be undone. Since the data structures

used by the runtime system to support properties are mostly C structures, and since the property information is stored in the capability structure and not in the object instance data itself, the shadow structures used by the Recoverable property cannot be used to recover all the state that existed prior to the method invocation.

To solve this problem, recoverable assignments which provide property inheritance are treated in a special way. When the assignment is made, only the minimal amount of work necessary to provide correct functionality is performed, and the remaining work is delayed until recoverable invocation terminates normally (or, in the case of a Dependent invocation on a Recoverable object, when the top-level invocation returns). This minimal amount of work involves modifying the property masks, and creating the data structures needed for any properties newly inherited by an object (e.g., creating locks for newly Controlled objects), but does not include destroying the data structures used by properties for any objects which were disinherited. Therefore, the locks, storage managers, shadow copies, etc. of objects which have been disinherited will be retained. Furthermore, the objects in the transitive closures will remain locked until the status of the invocation has been determined, as the objects must be protected against concurrent accesses throughout the time that their property sets might change.

One of two scenarios will now occur: either the invocation will terminate normally, or will it will be rolled back. After the minimal work described above is performed, a special function is queued up to be executed when it is known which of these two situations has occurred. This function is added to a list of similar functions kept by the `Thread` object in its `function_chain` instance variable. One function exists to handle the work associated with disinheriting objects and another to handle the work of bequeathing properties to objects. With the function, a list of parameters (which include the set of objects in the transitive closure) is kept, as well as the current level of the invoke depth (corresponding to the invoke depth of the current shadow copy). The execu-

tion of the functions in the `function_chain` is performed by the post-handler for the Recoverable property and by the code which implements the `restore` statement.

If the method terminates normally, the `function_chain` is traversed in the order it was created, and each function is executed in sequence. If a `restore` statement is encountered, the `function_chain` is traversed in the reverse order of its creation until the invoke depth of the functions in the chain are of a higher level than the invoke depth of the current recovery shadow. The functions take as one argument a boolean flag indicating the status of the method termination. This allows them to either complete the work begun when the assignment was made, or roll it back to the previous state it was in.

An alternate scheme to performing the minimal work necessary when a recoverable assignment is made is to make the optimistic assumption that the method will terminate normally, and do all the work associated with disinheriting and bequeathing properties accordingly. It is not possible to do a "pessimistic" assumption, since the object assigned to the instance variable must receive the services associated with the properties it inherited as a result of the assignment. However, this scheme will be very expensive in the case where the assignment is recovered, since all new data structures for the disinherited object will need to be created (which, in the case of Persistent or Durable objects, will require coordination with the on-disk storage; furthermore, the system must not in any case delete any on-disk versions until it is absolutely sure that the object is no longer Persistent or Durable). Delaying most of the work until later does not increase the amount of work that needs to be done. Furthermore, at least some work must be done even in the optimistic case, since all the objects in the transitive closure must be unlocked. Since the locks were acquired directly and not as the result of an invocation, it cannot be assumed that they will be released by the Concurrency post-handler, since the post-handler may never be executed.

3.5 Part-Of Clusters

The `partof` keyword also provides property inheritance, and therefore much of its implementation is similar to that described in Section 3.4. When the compiler detects an assignment to an instance variable marked `partof`, a special runtime function, `UpdatePartOf()`, is executed. As with property inheritance, this function and its support functions must treat recoverable assignments to Part-Of instance variables in a special manner, as described in Section 3.5.3.

In addition to providing property inheritance and dependency, the `partof` keyword is used by the runtime system to provide a logical clustering of objects. The transitive closure of all objects which can be accessed from a root object (i.e., an object which is not Part-Of any other object) using Part-Of references comprises the Part-Of cluster. This clustering information is currently used by the Durable and Persistent pre- and post-handlers to store and retrieve all objects in a cluster to and from disk as a single unit (see Section 4.3). Once object migration is implemented, clustering information can also be used to migrate entire clusters between disks. How objects are removed from and added to Part-Of clusters is described below. The effects of such changes to Part-Of clusters on object storage are discussed in Section 4.4.

There is one potential problem with using Part-Of clustering information in the manners described above. Consider the situation shown in Figure 6, where two objects form a Part-Of cluster. One thread (T1) invokes a method on one object of the cluster, while another thread (T2) invokes a method on the other. If T1 is a worker thread for a remote request the system may decide to migrate the cluster. However, `leafObject` may be in an inconsistent state because thread T2 is invoking on it. If the objects in the cluster are Controlled, the system can acquire all the locks before doing the migration; however, there is no guarantee that either `rootObject` or `leafObject` have concurrency control locks (and indeed, since only one thread is accessing each object at a time, neither object may need to be Controlled). The system has no mechanism by

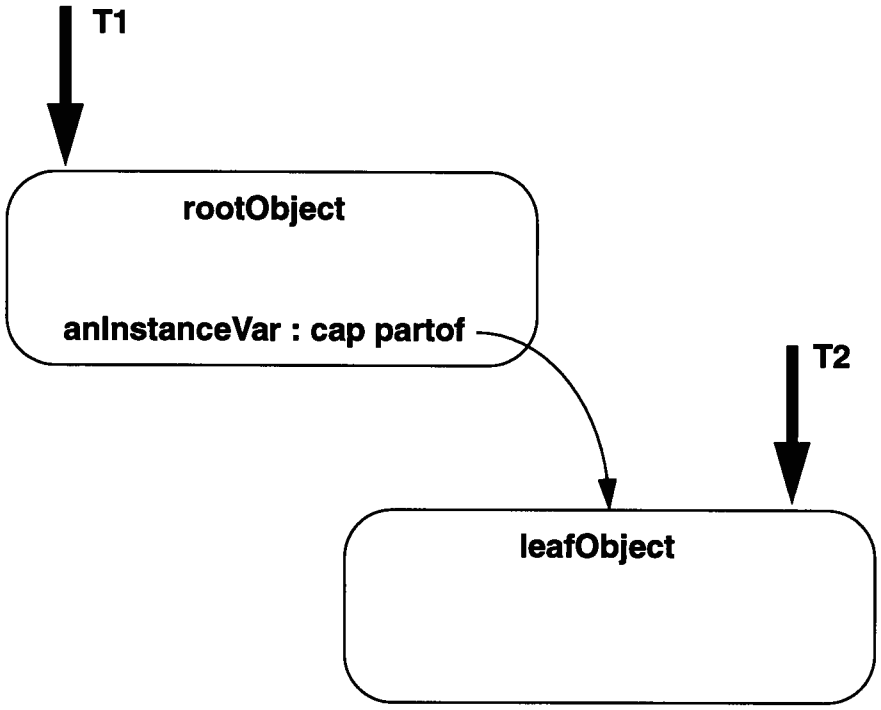

FIGURE 8. Objects in a Part-Of Cluster.

which it can detect that uncontrolled objects in a cluster are in a consistent state, yet it may need to make a copy of the instance data of these objects for the purposes of migration or object storage.

To solve this problem, and to more rigorously enforce the notion of clustering, a new concurrency control lock is added to each Part-Of cluster. This lock is shared among all the objects in the cluster, and any access to any objects in the cluster must first acquire this lock before acquiring any concurrency control locks on the objects in the cluster. Although this cluster lock renders any local object locks superfluous, the current implementation does not perform any optimizing and all locks are still acquired.

By placing a cluster lock on all the objects in a cluster, the system is effectively creating one Controlled object out of all the component objects in the cluster: only one thread may access the objects in a cluster at the same time. Although this scheme does solve the problem described above, adding the concurrency control lock will degrade performance slightly. Treating the object clusters as single objects has one additional drawback: it is not possible for a thread to spawn parallel invocations on the individual objects of a cluster, as deadlock will result.

The Part-Of cluster lock is acquired during the execution of the property pre-handlers, and is released during the execution of the post-handlers. The cluster lock is acquired just prior to and released just after any local concurrency control lock is acquired or released.

3.5.1 Removing Objects from a Part-Of Cluster

When an object is de-assigned from an instance variable which is marked `partof`, the system must first determine if the object is a single object or has `partof` references of its own. The transitive closure of all objects which inherit properties from the removed object is computed, and these objects are tagged to indicate if they simply inherit properties from the removed object or are Part-Of the removed object. These objects are locked, and the properties of the objects in the transitive closure are updated in the same fashion as described in Section 3.4. If the object has Part-Of references, then the removed object will form the root of a new Part-Of cluster. A new cluster lock is created and assigned to all the objects in the transitive closure which are Part-Of the root object.

Since Part-Of clusters have cluster locks, it's possible that a thread waiting on the cluster lock was attempting to invoke on the removed object (or, if the removed object is now the root of its own Part-Of cluster, one of the objects in the new cluster). Therefore, with each queued thread the identity of the target object of the thread is stored. When objects are removed from a Part-Of cluster, the cluster lock waiting list is examined, and any thread which is waiting to invoke on an

object no longer in that cluster will be moved to the new cluster lock (if one exists), or restarted if the removed object is no longer part of a cluster.

3.5.2 Adding Objects to a Part-Of Cluster

When adding an object to a Part-Of cluster, the system first ensures that the object is in the same address space as the cluster. As with removing an object from a Part-Of cluster, when adding an object the runtime system must determine if the object is the root object of its own cluster. If so, the objects in the old cluster must have their cluster lock pointers reset to point to the lock of the cluster to which they are being added. The threads waiting on the now unused cluster lock must be transferred to the new cluster lock. If the object being added is a single object (i.e., it has no Part-Of references) then any threads waiting on its local concurrency control lock (if one exists) are restarted. These threads are not allowed to begin execution of the method code, however; instead, they are required to begin the process of executing the property pre-handler code again. This forces them to acquire the new cluster lock before they are allowed to acquire the local concurrency control lock, for which they had been previously waiting. An alternate scheme to forcing a thread restart is to move all the threads queued on the local concurrency control lock to the cluster lock. The current implementation of locking makes this difficult, however, since the lock depth will not be correctly updated.

3.5.3 Recovering Part-Of Assignments

Because instance variables marked `partof` provide property inheritance, recoverable assignments to Part-Of instance variables are handled in the manner described in Section 3.4.1. As with any recoverable assignment that can affect multiple objects, the primary concern is that the objects be kept in a consistent state throughout the life of the invocation. Only a minimal amount of work is done when an object is added or removed from a cluster, and functions are queued up

on the `Thread` object's `function_chain` to finish the work begun, or undo the work started, when the status of the invocation is determined.

When a recoverable assignment is made, any objects removed from the Part-Of cluster retain their cluster lock. This allows them to remain protected against concurrent accesses while their status is in doubt. Dropping the cluster lock (or creating a new one if the removed object is the root of a new cluster) must be done anyway, and delaying this work until it is known that the assignment is permanent makes recovery easier, since any threads queued on the original cluster lock can be left untouched. If an existing Part-Of cluster is added to a Part-Of cluster, the existing cluster retains its cluster lock until the status of the invocation is determined. This lock is held by the thread which is performing the recoverable invocation, so all the objects in the cluster will remain in a consistent state.

3.6 Self-Invokes

The question arises about what to do about self-invokes, i.e., an object invoking a method upon itself. Self-invokes can be viewed in one of two ways, each with its own advantages and disadvantages:

(1) Self-invokes are considered to be "sub-invocations" of the initial object invocation. As such, no system services need to be provided for the invocation: all necessary pre-handlers were already executed before the initial invocation, and the post-handlers will be executed once the initial invocation finishes.

(2) Self-invokes are normal invocations, and are treated no differently than any other invocation. All the property pre- and post-handlers are executed for self-invocations.

If self invocations are viewed as sub-invocations, then the runtime can execute them more efficiently (since it does not need to check for object properties). Furthermore, the object can be continually modified by sub-invocations, but nothing will be written to disk until the initial invo-

cation returns. Any `restore` statement encountered will restore the instance data to the state it was in prior to the initial invocation; all work done inside the object to this point will be lost. This prevents an object from executing a transaction upon itself. It also raises the question of whether only invocations explicitly made on the object `self` should be considered sub-invocations, or if invocations on an instance variable which was set to `self` should also be sub-invocations. The runtime system has no mechanism by which it can determine which of these scenarios occurred, although the compiler could be modified to emit special code in the event of an invocation on `self`.

If self-invokes are classified as normal invocations, then redundant work will be done in some of the pre-handlers (e.g., the local concurrency control lock is obviously already held by the thread). Although some efficiency is lost, programmers are presented with a single, uniform model of object invocation, and they can view self-invocations as if they were any other invocation. A further question arises, and that is whether self-invocations should be considered Dependent or not. If they are considered Dependent, then the efficiency of the system can be further compromised, as extra work for Dependency must be performed; however, it does imply that writes will be delayed until the initial invoke returns. If self-invokes are not considered Dependent, then all invocations following the self-invocation would consider the self-invocation to be the root of their Dependent chain; this would preclude having self-invokes inside of transactions.

In the current implementation, all self-invocations are treated as normal invocations, and are furthermore considered to be Dependent invocations. Although the `dependentInvoke` flag is not set by the compiler in this case, the runtime system checks to see if the method invoker and invokee are the same object, and sets this flag if they are. Although this scheme is not the most efficient, it provides the most consistent support for Dependent invocations, and helps provide a uniform programming model.

3.7 Parallel Invocations and Property Support

As mentioned in Section 2.1, the Raven language contains special constructs to facilitate user-directed parallel programming. There are three primary forms of parallelism provided: companion invocations, early replies, and delayed results. The work on parallelism focused only on the Controlled property, in an environment that pre-dated Dependency. Most of the thought given to these features was therefore directed towards their impact on concurrency control [2]. Integrating the new property scheme (especially the notion of Dependency) with Raven's support for user-level parallelism has presented several challenges, many of which are yet to be fully resolved. For a further discussion of these issues, and alternate implementations to those chosen, see Chapter 5.

Property information is kept by the thread, so it is not possible to perform a Dependent invocation using a parallel thread of execution. There is no facility by which property information kept by one thread can be propagated to another thread, although some scheme such as the one used for remote invocations (see Section 3.3.3) could potentially be adapted. Consequently, multi-threaded transactions are currently not supported.

3.7.1 Companion Invocations

Companion invocations are invocations which are started from and executed in parallel with the current thread of execution. In the Raven language, executing the statement

```
!{ someObject.someMethod() }!.start();
```

creates a companion thread which will invoke the `someMethod` method on `someObject`. The target of the invocation (in this example, `someObject`) is treated as if it were not Dependent, even if the reference to the object is marked Dependent. If this object makes subsequent invocations on Dependent objects, they will form their own Dependent chain, and the target object the companion thread invoked on will be the root of the chain.

Another way to achieve parallel execution in a manner similar to a companion thread is for the user to simply create a new instance of the `Thread` class and have it invoke a method on an object. As with a companion thread, the initial invocation of this thread is treated as if it were not a Dependent invocation.

3.7.2 Early Reply

Early reply is a mechanism by which a method can return a result to its invoker but continue executing inside the method. The statement

```
result (value);
```

will return `value` to the caller, and a new thread will be created at this point to continue executing the method.

Early replies have severe consequences for the implementation of object properties, especially when the object is part of a Dependent chain. Consider a Controlled object in which an early reply is done. The initial invoker must give up the lock it acquired, and the spawned thread must acquire the lock. The question arises as to what happens if the object is in a Dependent chain. The invoking thread will not give up its lock, but the spawned thread will need to acquire the lock before it can do any work. Consider also a Recoverable object in which an early reply is done. It is unclear what should happen if a `restore` statement is encountered by the new thread executing after the result has been returned: should the object be restored to the state it was in when the `result` statement was executed, or to the state it had prior to the initial method invocation? What if the object is part of a Dependent chain, and an object above it in the chain executes a `restore`? How can work done by the spawned thread be restored properly, especially if the thread is still executing inside the object? Similar problems relate to the Durable and Persistent properties: should the object state be written to disk at the point of the `result`, when the spawned thread terminates, or both? If only at the point of the result, then changes made by the

spawned thread could be lost; but if the object is Durable and part of a Dependent chain, then the object state cannot be written by the spawned thread when it terminates, since the object state can only be written along with the objects comprising the initial Dependent chain.

Rather than attempt to resolve all these issues, and to simplify the implementation, early replies are ignored in some circumstances. If the object is part of a Dependent chain, or has the Recoverable, Durable, or Persistent properties, then the `result` statement will simply save the value to be returned to the caller and use it as the return value when the method terminates (ignoring any subsequent value specified by a `return` statement as the result to return). A new thread is not created; instead, the current thread continues execution, and control is not returned to the caller until an actual `return` statement is encountered or execution of the method finishes. This choice decreases the parallelism of the system, but ensures its correct operation when an early reply is encountered.

3.7.3 Delayed Result

Delayed result allows the execution of a method to finish with the expectation that a different thread will actually provide the result to return to the caller. The statement

```
leave;
```

acts like a `return` statement, except the value to be returned to the caller is not taken from the body of the method but is instead sent to the current thread by another thread. If this value hasn't yet been sent, the current thread blocks (and control is not returned to the caller) until the value is received. A thread sends a return value to another thread using the `result` statement:

```
result thread value;
```

sends `value` as the value to return to the thread `thread`.

To properly implement delayed result, any locks held on the object (both concurrency control locks and cluster locks) must be released when the `leave` statement is executed. This is because it may be necessary for another thread to enter the object in order to perform the `result` which will send the value to the waiting thread. However, if the object is in a Dependent chain, locks are not supposed to be released until the top-level invocation completes. If the requirements of Dependency are maintained, then using a delayed result may bring the system into a state of deadlock; if locks are freed, then the semantics of Dependency are not preserved.

Since delayed result provides programmer-specified parallelism, the current implementation will release any local locks when a `leave` statement is executed, even if the object is in a Dependent chain. It is assumed that the programmer will take into account that locks will be released when writing code that executes a `leave`, and will not attempt to use such objects in transactions, since a significant consequence of releasing the locks is that serializability will be lost when another transaction invokes on the object while a thread is suspended waiting for a result.

Releasing local locks when a `leave` is encountered creates the potential for an additional problem. Consider a thread T which is suspended in an object O waiting for a result to return to its caller. It's possible for another thread T' to acquire the locks on O and assign O to an instance variable marked inherits, or de-assign O from such an instance variable. When T resumes, the properties of O have changed from what they were when the invocation on O began. Property post-handlers are executed after a suspended thread resumes, creating a potential mismatch in the execution by T of the property pre- and post-handlers. The current implementation therefore generates a runtime error if an object which contains suspended threads is assigned to an instance variable marked `inherits` or `partof`.

3.8 User Properties

User properties are supported in the same way as the system properties, through the execution of pre- and post-handlers. The invoke routines and the system pre- and post-handlers are all written in C. Allowing programmers to write their pre- and post-handlers in C as well is possible, but is fraught with problems. If users are allowed to write in C, they can become tempted to modify system data structures. Furthermore, programmers would need to know details of the Raven implementation in order to access object instance data; however, these details may change, introducing bugs into the user code.

Given these considerations, user pre- and post-handlers are therefore supported by performing invocations on the object. The basic `Object` class supports four (empty) pre-methods and their corresponding post-methods:

```
behavior preUserN(dependentInvoke: Int);
behavior postUserN(dependentInvoke: Int, isDependentRoot: Int, hasProp: Int);
```

Where *N* can be either `1`, `2`, `3`, or `4`.

The integer parameter `dependentInvoke` which is passed to the user pre- and post-handlers is set to either `True` or `False` (predefined boolean values supported by the Raven language) depending upon whether the current object is Dependent upon its invoker. Similarly, the parameter `isDependentRoot` will be `True` or `False` depending upon whether the current object is the root object of a Dependent invocation chain. This information is provided to the programmer so that user properties can also be designed to take advantage of Dependent invocations. In such circumstances, the programmer may need to subclass the `Thread` class so that state information for the user property can be kept in the thread along with the information for the Controlled, Recoverable, and Durable properties. The integer parameter `hasProp` is set to `True` or `False` depending on whether the current object actually has the user property. This flag is neces-

sary since the post-methods will be executed when `isDependentRoot` is `True`, even when the object does not have that property. Without this flag, the programmer would have no way of determining if the root object of a Dependent invocation chain actually had one of the user properties.

The pre- and post-methods will be invoked on an instance which has user properties. When a programmer wishes to implement a user property for a class, the programmer must override these methods in the class. The invocations of the pre-methods must be done in such a way that they do not result in a continual recursive execution of the pre-methods; similarly, the post-methods must be executed in such a way that they don't cause the pre-methods to execute. This is done by calling the method code directly, bypassing the property support routines. The pre-handlers are only executed on the machine where the object resides, so there is no worry that the object data will not be local.

Although self-invokes are normally considered dependent invocations, the pre- and post-methods are not considered dependent invokes. Since the user pre-methods are executed after the Recoverability pre-handler, any `restore` statement executed inside the object method code will restore the object to the state it was in prior to the execution of the pre-methods. This may create some difficulties, since the current implementation does not re-execute the pre-methods (and the post-methods will be executed when the object invocation finishes). The issues surrounding user properties are further explored in Chapter 5.

Since the pre- and post-handlers are written by the user, the system cannot make any guarantees about the orthogonality of user properties. No tools have been written to test for such orthogonality.

3.9 Status of the Implementation of Properties

Although semantics have been developed for all the properties (see Section 2.4), the runtime system does not yet provide all the services for which there are properties. The properties which are currently implemented are the following:

- Controlled
- Recoverable
- Persistent
- Durable
- Immutable
- User properties (as described in Section 3.8)

Details of the implementation of the Controlled property can be found in [1]. Details of the implementation of the Recoverable property can be found in [7]. Details of the implementation of the Persistent and Durable properties can be found in Chapter 4.

Dependent invocations are supported by the Controlled, Recoverable, Persistent, and Durable properties, even when such invocations are remote. Although the Raven semantics say nothing about Dependent Persistent invocations (Section 2.6.1), the implementation treats them in the same manner as objects which are Durable. Dynamic property inheritance, through both the `inherits` and `partof` keywords, is fully supported. This includes the case when the assignment to an instance variable marked `inherits` or `partof` is recoverable: if a `restore` statement is encountered in the current Dependent calling chain, the object property sets will be correctly restored, and all data structures (such as lock structures) used in the implementation of system properties will be correctly updated. However, the system only allows property inheritance from local objects (i.e. objects which are in the same address space).

CHAPTER 4

Object Storage: Details of the Durable and Persistent Properties

4.1 Object Storage Overview

Objects in Raven are normally volatile; they exist only in memory, and persist only as long as the active process which created them (or until they are garbage collected [1]). When an object is given either the Persistent or Durable properties, the Raven system will ensure that a copy of the object's instance data will be written to disk. Since the Persistent and Durable properties function in almost identical ways (see Section 2.4 for a description of their semantics), the concepts discussed in this chapter apply equally to both properties. The only differences between the two properties arise in their implementations, and are described in Section 4.3.5. In this chapter, objects which are described as being *storable* are meant to be those which have either the Persistent or Durable properties.

The object storage system provides the following features:

- *Reference swizzling*: All references to other objects are capability pointers, which point to the location in memory where the object's capability structure resides. All references to an object must be converted to unique identifiers which correspond to the object. This allows the system to reload the object at any position in memory, as the unique identifiers can be translated back to the new capability pointer.
- *Whole object storage*: Objects must be stored in their entirety in an atomic fashion. If the object's state isn't written atomically to disk, a failure during the disk write could result in an inconsistent version of the object on disk, and subsequently an inconsistent version of the object will be loaded into memory.
- *Lazy loading of object data*: When a reference to an object is unswizzled, if that object is not currently in memory, the runtime system will not fetch the object data immediately. The object's instance data will be fetched only when it is actually needed, i.e. when an invocation is made on the object.
- *Object clustering in storage*: Objects which comprise a Part-Of cluster are stored together on disk, and loaded together from disk.

Although the Raven runtime system provides garbage collection for in-memory objects, no garbage collection is currently provided for objects in storage. The system will remove an object from storage when it is no longer a storable object (e.g., while an object inherits the Persistent property it will be stored to disk, but when it no longer inherits that property it will be removed from disk storage).

4.1.1 Globally Unique Identifiers

To implement reference swizzling, the system must provide a globally unique identifier, or *GID*, for every storable object. `GIDs` are also useful for accessing remote objects, but in the absence of object mobility and persistence the object's location information can be used to provide a unique identifier. The original version of Raven used location information as the global identifier, and was therefore unsuitable as a true globally unique identifier.

The actual GID of an object is a 128-bit value composed of four distinct 32-bit fields (see Figure 9). The first field contains the IP address of the machine on which the object was created. The second field contains the port number to which the Raven World which created the object was attached. The third and fourth fields are incremental counters. The third field is incremented once upon each startup of a Raven World, and the fourth is incremented once for each GID assigned by the World. The fourth counter is reset to zero upon each startup of a World.

```
{
u_long          creator;
u_long          world;
u_long          generation;
u_long          name;
}
```

creator:	IP address of creator machine
world:	Port number of Raven World
generation:	Raven World incarnation counter
name:	Per-incarnation GID assignment counter

FIGURE 9. GID Structure.

A GID is assigned to every storable object. An object which is referenced from another Raven World (i.e., for which a Proxy object [1] exists in the other World) is also assigned a GID. GIDs are not normally assigned at object creation time unless the object is storable. The object's capability structure (see Figure 6) contains a pointer to a structure which in turn contains a pointer to the GID structure (see Appendix A.6). If the object does not have a GID, the pointer in the capability structure will point to NULL.

4.2 Storage Model

The basic storage model used is outlined in Figure 10. At the highest level are the Raven objects themselves, which exist in main memory. Each storable object has an associated *Storage Manager* object (see Figure 11), which is an instance of the `StorageManager` class (see Appendix B.2). The storage manager is responsible for overseeing the storage of the object's instance data to disk, as well as converting stored data into user objects.

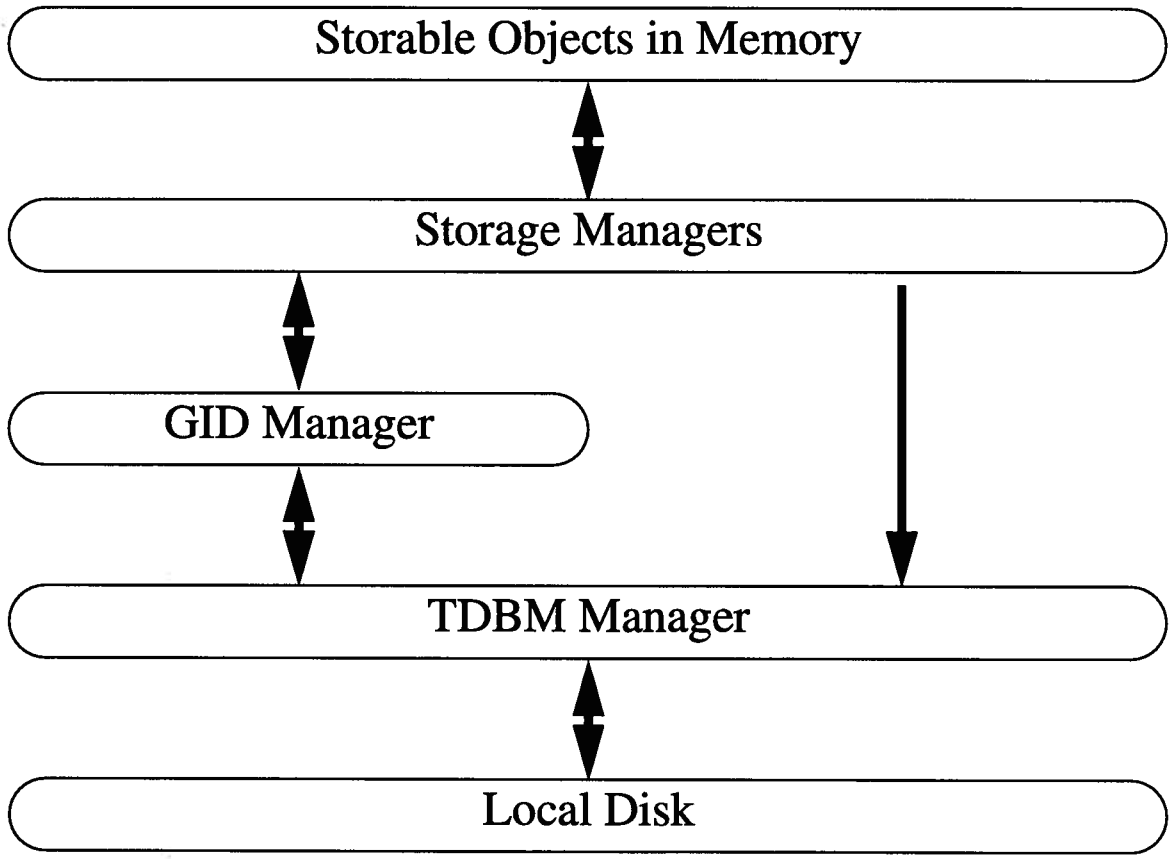

FIGURE 10. Object Storage Model.

FIGURE 11. Storage Manager Role.

At the lowest level is the local disk itself. To achieve whole object storage, TDBM [6] is used to write the object state to disk. TDBM is a transaction oriented database that stores key/value pairs. TDBM will ensure that all the data presented to it is written in an atomic fashion to a local disk. Although multiple Raven Worlds on the same machine may use the same disk, each World has its own *TDBMmanager* object (see Figure 12), which is an instance of the `TDBMManager` class (see Appendix B.5). The TDBMmanager object is accessible inside Raven applications as the global object identifier `TDBMmanager`. The TDBMmanager object writes the data for all the objects in the Raven World to a file specific to that World. If the Raven World is shut down and a new instance of Raven is launched with the same World identifier (i.e., on the same machine using the same port number as the previous World), it is considered the same World and will use the same file.

TDBM cannot guarantee atomic updates of data if the database file is located across NFS. Object storage should therefore be done on a disk physically mounted on the same machine on which the Raven World is running, or the system will be vulnerable to NFS failures. The local directory under which TDBM should store its files is determined at startup time by examining the environment variable `RVSTORAGEPATH`. If no such environment variable exists, a default value is used; however, this default value is not guaranteed to exist on any particular host.

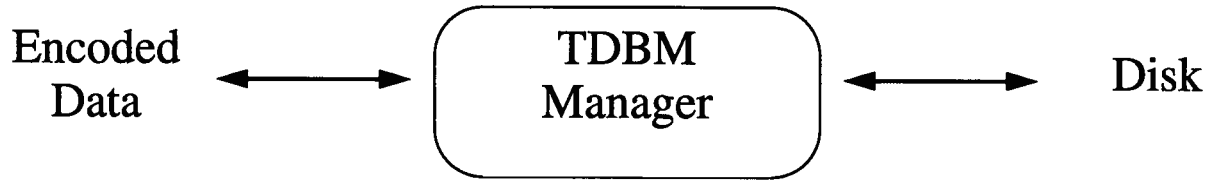

FIGURE 12. TDBM Manager Role.

Each Raven World has an associated *GIDmanager* object (see Figure 13), which is an instance of the `GIDManager` class (see Appendix B.4). The GIDmanager object is accessible inside Raven applications as the global object identifier `GIDmanager`. The GIDmanager has several responsibilities:

- It assigns GIDs to storable objects.
- It maintains a GID to capability map. For every (currently known) object with a GID, this map provides the current location in memory of the capability structure for the object. This map is necessary during unswizzling, when GIDs must be converted to capability pointers.
- It creates capability structures for objects loaded in from disk.
- It intercepts invocations made on objects currently not in memory and interacts with the TDBMmanager to fetch the objects from disk.

4.2.1 Object Storage Events

After an invocation which modifies the instance data of a storable object completes, the object's instance data must be written to disk. This involves the following actions:

(1) The object's storage manager is notified that an invocation occurred on the

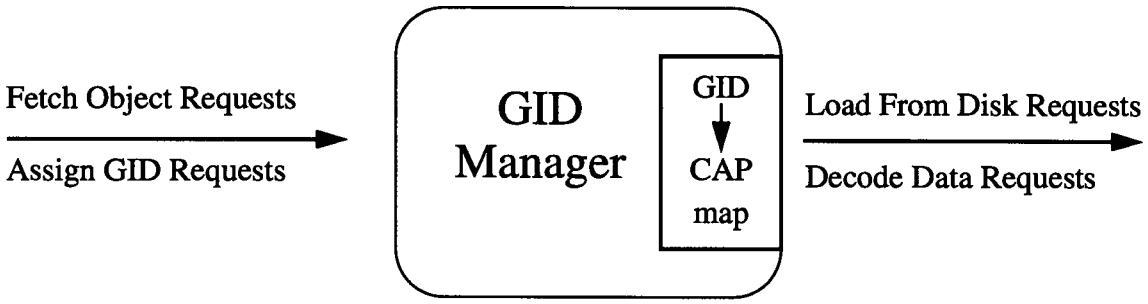

FIGURE 13. GID Manager Role.

object.

(2) The storage manager instructs the object to encode its instance data.

(3) The storage manager adds the information in the object's capability structure to the encoded data.

(4) The encoded data is passed to the TDBMmanager along with the GID of the object, which is used as the key under which the data is stored.

(5) The TDBMmanager writes the data to disk.

4.2.2 Object Loading Events

When an invocation is made on an object which is not in RAM, the object must be loaded in from disk. The capability structure used for objects not in RAM is a "skeleton" capability structure: the only fields of the structure which are filled in are the `GID` and the invoke routine (see Figure 6). The invoke routine used by skeleton capabilities is a special routine called *`FetchInvoke`*. `FetchInvoke` ensures the object data is loaded into disk and that the skeleton capability struc-

ture is filled in before allowing the invocation to proceed. The exact sequence of events involves the following actions:

(1) The `FetchInvoke` routine calls the GIDmanager, and asks it to load in the data for the object with the GID found in the skeleton capability structure.

(2) The GIDmanager passes the GID of the object to the TDBMmanager.

(3) The TDBMmanager fetches the encoded data stored under that GID from disk.

(4) The GIDmanager extracts the capability information and fills in the skeleton capability structure, including replacing the `FetchInvoke` function with the appropriate invoke function for the object.

(5) The GIDmanager creates a storage manager for the object, and passes the encoded data to the storage manager.

(6) The storage manager creates an object of the appropriate class, and has it load its instance data using the encoded data.

(7) During decoding, all `GIDs` are passed to the GIDmanager to be unswizzled and turned into capability pointers. If no capability exists for the `GID`, the GIDmanager creates a skeleton capability structure for the object.

4.3 Implementation Details

The StorageManager objects, GIDmanager, and TDBMmanager are all Raven objects and interact with each other using normal object invocations. Much of the implementation of these objects, however, is directly in C, although the Raven language has been used where possible. As an object's instance data is encoded and the capability information added to form the complete encoding of the object, Raven `Integers` are used to hold pointers to the data for passing between a Storage Manager, the TDBMmanager, and the GIDmanager.

4.3.1 Object Encoding

When a Storage Manager needs to get the encoded form of an object, it asks the object to encode itself and return the encoded data. Similarly, a Storage Manager will provide an object with a pointer to encoded data, and have the object load its instance data using the encoded data. The `Object` class provides two basic methods:

- `behav enCodeData() : Int`

This method traverses the object's instance data and writes an encoded version of the data into memory, swizzling all object references and encoding primitive integers and floating point numbers (see below). A pointer to the location of the data is returned.

- `behav deCodeData(data : Int);`

This method replaces the object's current instance data with the data obtained by decoding the data pointer passed in.

The format of a buffer containing encoded data in memory is shown in Figure 14. The buffer begins with an integer representing the total length in bytes of the buffer (including the four bytes needed for the integer), followed by the encoded instance data.

FIGURE 14. Format of encoded buffer in memory.

Raven currently supports three different types of instance variables: `Int` (simple integer), `Float` (floating point number), and `cap` (object reference). While encoding (or decoding) an object, the type of instance variable is determined by examining the definition of the object's class.

`Ints` are stored as instance variables in their (usually) 32-bit machine-dependent representation [1]. This allows integer operations to be efficient. They are encoded by simply copying them directly into the encoded buffer. Conversion into an external representation (such as that obtained using XDR [25], which is currently used to encode requests and replies for remote invocations) is not needed nor done, since the object will always be reloaded into the current Raven World, and the assumption is made that the machine representation of integers will not change between incarnations of a Raven World.

Like `Ints`, `Floats` are also stored as instance variables in their machine-dependent form, and are encoded by simply copying the floating point number into the encoded buffer.

Object references are swizzled by writing out the `GID` of the object to the encoded buffer. Upon decoding, the `GID` is converted to a local capability by the GIDmanager. If the object referenced is not storable, zero values are written to the encoded buffer in place of the actual fields of the `GID` (which may not even exist for a non-storable object). Since a non-storable object will not survive system restarts, if an actual `GID` were stored the encoded object would contain a reference to an object which no longer exists. Upon decoding, a zero-valued `GID` is treated as a reference to the `nil` object. This practice precludes the use of the object storage system as a mechanism for providing object paging.

The current implementation relies upon the fact that each of the values stored in the encoded buffer are a multiple of four bytes. If the encoded buffer itself is allocated on a four byte boundary (true for the current implementation of the Raven `MALLOC ( )` macro), then the values (especially

integers and floating point numbers) can be easily written to and extracted from the buffer. Since class definitions can vary widely, the size and format of different encoded buffers is highly variable, and the object class must be known to properly decode an object's data.

The data in the encoded buffer contains only the object's instance data and lacks other important information about the object which is contained in the object's capability structure. In addition to the instance data, an object's class, property set, parent, and inheritance root must also be stored. Storing this information allows the capability structure to be rebuilt upon object loading, and provides the class information necessary for decoding the data. Figure 15 shows the layout of the complete encoding of an object, including the capability information. The class name, which is assumed to be unique for all classes (an assumption shared by the remote invocation mechanism [1]), is stored along with its length. If necessary, padding is added after the class name to ensure that the properties field begins on a four byte boundary. The properties stored in the encoded form of an object may be a different set than those the object currently has. This is because the object can inherit properties from a non-storable object, in which case the property mask to be stored must be calculated by traversing up the inheritance tree until the top-most storable object from which properties are inherited is found. The Parent `GID` field holds the `GID` of the object's parent (i.e., the object from which properties are directly inherited). The Inh_root `GID` field contains the `GID` of the top-most storable object in the inheritance tree. The Instance Data field contains the contents of the encoded buffer shown in Figure 14. A final pad, if needed, is placed at the end to ensure the entire encoded object is a multiple of four bytes. Since the encoded buffer will always be a multiple of four bytes, and the class name with its pad will be a multiple of four bytes, there should never actually be a need for a trailing pad.

The process of adding the object's capability information to the encoded data is done by the object's Storage Manager. The GIDmanager is responsible for extracting the capability information and filling in the object's capability structure when an encoded object is loaded from disk.

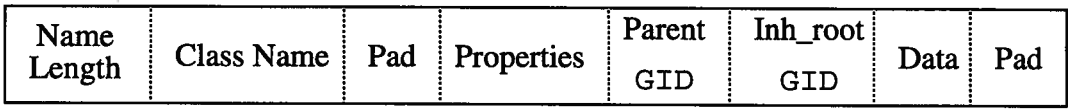

FIGURE 15. Format of a fully encoded object.

4.3.2 Storage Manager Implementation

Objects maintain a pointer to their storage manager inside their capability structure. When a storable object is created, or when an object inherits the Persistent or Durable properties, the object will be assigned a Storage Manager. All of the storable objects in a Part-Of cluster each use the same Storage Manager.

Storage Managers maintain a "cached" version of the encodings for every object they manage. These caches serve two purposes:

(1) They allow the Storage Manager to compare the object's current state with its previous state, to determine if the object state has actually changed. If the object is unchanged, there is no need to write the object to disk, thus saving significant performance overhead.

(2) They allow the Storage Manager to submit all the objects in a Part-Of cluster to the TDBMmanager for storage at once. If the Storage Manager didn't maintain cached versions, it would have to encode and encode each object in the cluster every time it needed to write the cluster to disk. Furthermore, if the Part-Of cluster lock is ever abandoned, a lack of cached versions could force the Storage Manager to acquire a lock on every object in the cluster before encoding each object. No cluster lock implies multiple threads could be inside the cluster, so the Cluster Manager may have to wait a significant amount of time in addition to increasing the likelihood of deadlock.

The encoded data of an object is kept by a Storage Manager inside an object of the `SMEntry` class (see Appendix B.3). An SMEntry object contains the object capability pointer, a pointer to the object's encoded data, and the length of the encoded data. If a Storage Manager manages multiple objects, it keeps the SMEntry objects in a dynamically sizeable array. The SMEntry for the root object of the Part-Of cluster is always the first object in the array.

4.3.3 TDBMmanager Implementation

A Storage Manager presents the encoded data streams to the TDBMmanager for writing to disk. The encoded data is packaged into an object which is an instance of the class `TDBMDatum` (see Appendix B.6). A TDBMDatum object is an object encapsulation containing two of the `TDBMDatumStruct` data structures used by the TDBM library. It contains four integer values: one integer value used as a pointer to the data, a second holds the data length, a third is used as a pointer to the key under which the data is to be stored, and the fourth contains the length of the key. A TDBMDatumStruct also requires alignment information, but data and keys are always assumed to be aligned on a four byte boundary. When a Storage Manager invokes on the TDBMmanager, it creates a TDBMDatum object and sets the data pointer to point to the encoded data stored in the SMEntry object. A pointer to the object's `GID` is used as the key under which the object is to be stored. If the Storage Manager manages multiple objects, it passes an array of TDBMDatum objects to the TDBMmanager. Each TDBMDatum object occupies the same position in the array passed to the TDBMmanager as the SMEntry object from which the TDBMDatum was formed occupies in the array kept by the Storage Manager. As such, the TDBMDatum object containing the encoded data for the root object of the Part-Of cluster will be the first object in the array passed to the TDBMmanager.

The TDBMmanager stores data to disk in one of three formats, which are shown in Figure 16. The basic structure of all three formats is `count | data`, where `count` specifies the number of objects stored in the `data`. The key under which the data is stored is an object `GID`.

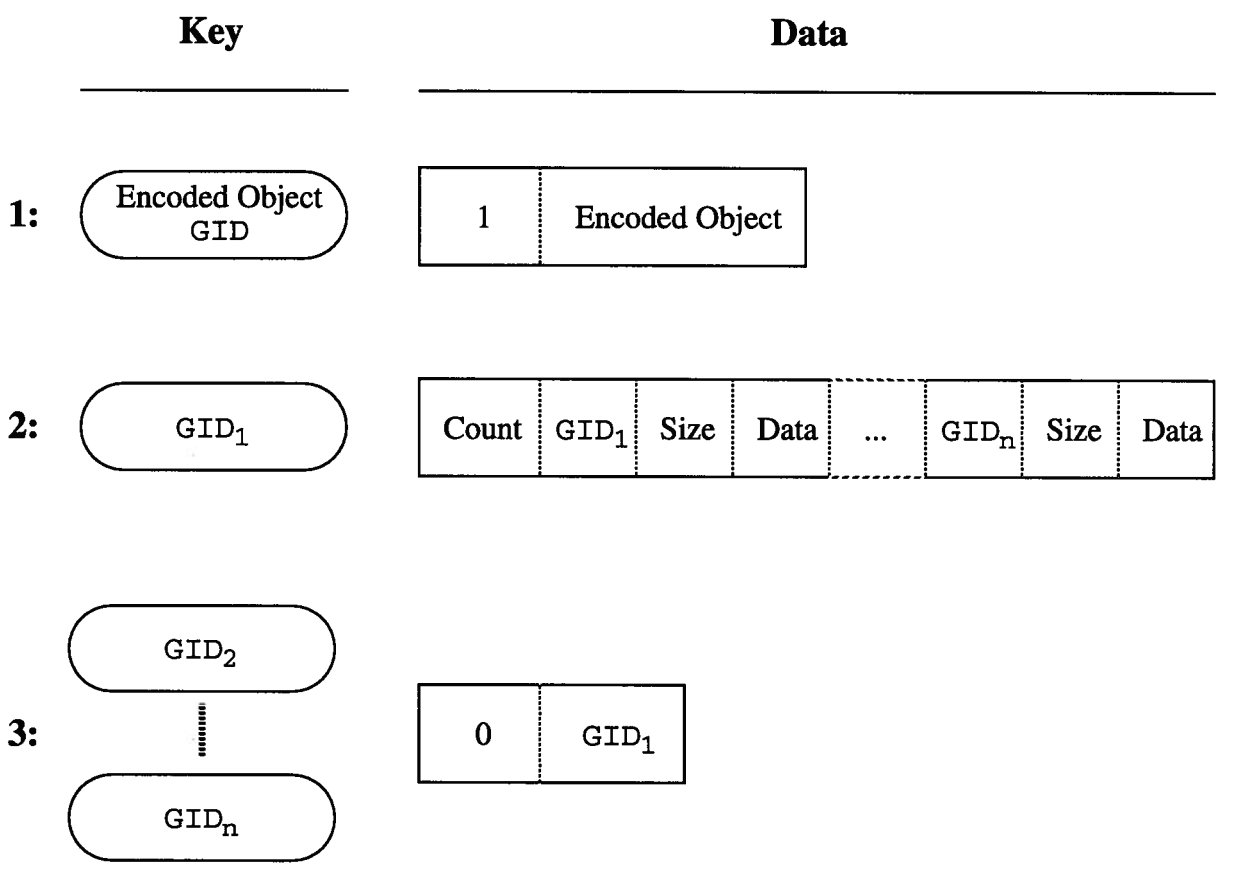

FIGURE 16. Formats of data storage on disk.

Format type 1 is the most basic, and shows how a single TDBMDatum object which is presented to the TDBMmanager will be stored. The `count` field is one, since only one object is stored. The `data` field contains the encoded data that was passed in the TDBMDatum object. The key under which this data is stored is the key contained in the TDBMDatum object, which is the `GID` of the object.

When the TDBMmanager is presented with an array of TDBMDatum objects, it stores all the objects together on disk in one continuous block, as shown in format type 2. In this case, `count` contains a count of the number of encoded objects stored in the block, which is equal to

the number of TDBMDatum objects in the array. It is followed by <count> tuples consisting of the key under which the object was to be stored, the size of the object's encoded data, and the encoded data itself. The entire block is stored using the key found in the first TDBMDatum object of the array, which will be the `GID` of the root object of the Part-Of cluster which is being stored. Storing the size of the encoded data allows the TDBMmanager to quickly transform a block of clustered objects into individual TDBMDatum objects, one for each stored object, without having to know the format of the encoded data.

When an array of TDBMDatum objects are to be stored, all the encoded objects will be stored using the `GID` of the root object as the key. To allow the system to locate the other objects in the cluster, the TDBMmanager stores a data block of format type 3 under the `GID` of these objects. The `count` field is set to zero, indicating that the data is the actual key under which the object's data is stored.

When an object is to be loaded in from disk, the TDBMmanager is presented with a TDBMDatum object with only the instance variables corresponding to the key set. The TDBMmanager uses the key information to load in the associated data from disk. The data is examined to determine which format it is in. If the data block is of format type 3, the TDBMmanager uses the stored key to fetch the actual data. If only a single object was stored under the key, the TDBMmanager returns the passed in TDBMDatum object with the integers for the value pointer and size set to the encoded data. If multiple objects were stored in a cluster under the key, an array of TDBMDatum objects is returned. If no object is stored under the key presented, the TDBMmanager returns the `nil` object.

Because Raven is multi-threaded, the TDBMmanager has the Controlled property to ensure that accesses to the disk are serialized and that the whole system does not enter a blocked state when a particular thread is blocked waiting for a lock from TDBM itself.

4.3.4 GIDmanager Implementation

The GIDmanager interacts with the TDBMmanager to load encoded objects from disk. If the TDBMmanager returns `nil` to the GIDmanager, the GIDmanager attempts to find the object somewhere on the network. The `creator` and `world` fields of the object's `GID` are examined, and a remote invocation is made on the GIDmanager of the Raven World at that location in an attempt to find the object. When object migration is implemented in Raven, a broadcast or other mechanism will be needed to find the actual location of the object, since there will be no guarantee that an object will remain in the World in which it was created.

If the TDBMmanager returns encoded data to the GIDmanager, the GIDmanager begins the process of unencoding the object data. First, it creates a Storage Manager object to manage the storage of the object (or objects, if a cluster of objects were returned by the TDBMmanager). Next, it fills in the capability information for each object, creating a capability structure first if necessary. Any necessary locks (including a cluster lock) are created. The encoded data is then passed to the Storage Manager so the object instance data can be decoded.

As described in Section 4.2.2, capability structures that exist for objects not currently in RAM use a special invoke routine called `FetchInvoke`. When a `FetchInvoke` is performed, the first action of the GIDmanager is to ensure that the object data has not already been loaded in. Although the GIDmanager is controlled against concurrent accesses, multiple `FetchInvokes` could be performed on an object before the first has completed. The final action of the GIDmanager is to set the invoke routine to the normal invoke to be used for the object.

The GIDmanager also maintains a mapping of `GIDs` to capabilities. This allows it to find the local capability pointer for any object based upon its `GID`. The map is simply a hash table that associates the four integer fields of the `GID` with the capability pointer of the object. `IIIIISc` [10], a table package which performs {4 integer}→{1 integer} mappings, is used to manage the

hash table. If the GIDmanager is asked for the capability pointer for a particular `GID`, and no such capability currently exists, the GIDmanager creates a skeleton capability structure whose invoke routine is `FetchInvoke`; it then returns a pointer to this structure. The GIDmanager will turn this capability structure into a `Proxy` object if the object cannot be found on the local machine when the GIDmanager attempts to load the instance data of the object.

4.3.5 Implementation Differences Between the Persistent and Durable Properties

In the current implementation, the Durable and Persistent property handlers are almost entirely identical. Objects have only one Storage Manager, which acts for either the Durable or Persistent properties (or both, if an object actually had both properties). In the case of Dependent invocations, both properties are handled in the same way, and conform to the Durable notion of storage: control will not be returned to the caller until all the objects have been written to disk. The only current difference between the implementation of the two properties is that in the case of Persistent objects, a new thread is spawned to perform the work of storing the object to disk, allowing the original thread to return immediately. This does provide faster performance for the user, but does not provide any I/O optimizations that might be available by delaying writes to disk, such as allowing some updates to be done only in RAM. This can be accomplished with an active thread inside each Storage Manager which periodically writes the cached data to disk.

4.4 Dependent Invocations on Storable Objects

When storable objects are part of a Dependent chain, the Raven system must delay writing the objects to disk until the top-level invocation completes. Furthermore, the objects must be written atomically; i.e., either all the objects get written to disk, or none of them do.

To support Dependent invocations on storable objects, each Thread object keeps a list of the Storage Managers for the storable objects it invokes upon in a Dependent chain. A pointer to this

list is kept in the Thread's `storage_chain` instance variable. Each entry in the storage_chain (see Figure 17) is a structure which includes the `dependentID` of the Dependent chain, the `cap` of the Storage Manager, and a list of the objects which are managed by that Manager and which have been invoked upon during the current Dependent invocation chain. The Durable and Persistent post-handlers append the current invokee's Storage Manager to this chain. If the Manager is already on the chain, the current invokee is appended to the list of modified objects kept for the Manager.

```
struct STORAGE_INFO
{
    struct STORAGE_INFO     *next;
    struct STORAGE_INFO     *previous;
    int                     dependentID;
    cap                     manager;
    LIST                    *objects;
};
```

FIGURE 17. Format of `struct STORAGE_INFO` data structure.

Normally, the TDBMmanager expects to perform the storage for a single Storage Manager as one transaction. In a Dependent chain, however, multiple Storage Managers must have their data stored by the TDBMmanager using the same transaction identifier. To accomplish this, the following steps are taken when the top-level invocation of the Dependent chain completes:

(1) A transaction identifier is acquired from the TDBMmanager.

(2) For every Storage Manager in the `storage_chain`, the local cache of each object in the `objects` list for that Storage Manager is updated.

(3) The Storage Manager is directed to write the list of cached objects to storage,

using the transaction identifier acquired in step 1.

(4) The TDBMmanager is directed to commit the transaction associated with the transaction identifier.

4.4.1 Remote Dependent Invocations on Storable Objects

The semantics for Dependency specify that all the objects in the Dependent chain will be stored together. This requires that when Durable objects in a Dependent chain are in different Raven Worlds, a two-phase commit protocol (or similar protocol) be used to ensure that all the objects are written to disk atomically.

In the current implementation, a two-phase commit protocol is not implemented, as it is beyond the scope of this thesis. Instead, each of the Raven Worlds involved in a distributed transaction will write their data separately from the other Worlds, although atomically within each World.

4.5 Storage of Collection Class Objects

The Raven class library supports a variety of classes which are collections of other objects. These classes all inherit from the `Collection` class, and include the `Array`, `MemoryArray`, `List`, `Set`, and `String` classes (see [1] for a full description of the Raven class library, including the `Collection` classes). The implementations of these classes often rely upon other classes (usually an instance of `MemoryArray` at the lowest level). For example, the `String` class uses an `MemoryArray` object to store the actual characters of the string.

This implementation presents a problem for object storage. Consider a Persistent String object. If this object is encoded normally, the actual characters of the string will never be stored, as these actually reside in the MemoryArray inside the String. In fact, although the String itself is Persistent, the MemoryArray is not, and so the encoded data will contain a reference to the `nil`

object and not even to the MemoryArray which contains the characters. Similar problems exist for the other `Collection` classes. `MemoryArray` has its own storage problem: its implementation is done directly at the C level, so normal Raven statements cannot be used to swizzle an instance of `MemoryArray`.

One solution is to attempt to use the Part-Of or Inherits references when designing the `Collection` classes. For example, the `String` class can indicate that the MemoryArray object should be Part-Of the String, so if the String is Persistent then the MemoryArray will be also. There are several problems with this approach, however. If Part-Of is used, then the `Collection` class object will be given a concurrency control lock, which will decrease the performance of operations on the object. If Inherits is used instead, then the object will be stored in pieces on disk, and multiple fetches will be required to load it into RAM. A more fundamental problem is that although this scheme will work for the `String` class, it will currently not work for any of the `Collection` classes which are implemented by composing multiple `Collection` class objects together. The `Array` class, for example, is implemented as a multi-way tree, using objects of the class `FixedArray`. At any given level, a FixedArray may contain elements of the Array or another FixedArray. Although the top level FixedArray used by the `Array` class can be specified as being Part-Of the Array, there is no mechanism by which other FixedArrays placed into the top level FixedArray can be declared as being Part-Of also. Therefore, the Part-Of tree stops after one level, and parts of the Array will not be Persistent. One simple solution is to re-implement the `Array` class (and the other `Collection` classes) in such a way that Part-Of can be used so all components of the class can inherit properties. Indeed, when a class is written all objects which are used to implement the class should be defined as Part-Of. Another solution is to use the techniques described in Section 5.4 to implement the `Collection` class objects, which would allow the Part-Of relationships to extend downward as many levels as necessary to include all the component objects of the `Collection` class object.

To allow `String` objects to be properly stored, the `String` and `MemoryArray` classes override the `enCodeData` and `deCodeData` methods found in the `Object` class with special versions which will properly encode and decode the contents of a String or MemoryArray object. This scheme can be adopted for all the different `Collection` classes, although at present the `enCodeData` and `deCodeData` methods have been overridden only in the `String` and `MemoryArray` classes.

4.6 Object Name Service

Once object storage has been implemented, some mechanism by which users can identify and name objects must be provided, so that after a system reboot particular objects that have been stored to disk can be reloaded. Otherwise, the user will never be able to find the objects stored to disk. Using ASCII (string) names is a natural convention for naming objects (as they are for naming Unix files), and much less cumbersome for users than requiring that they remember the `GIDs` of the objects which they are interested in. A rudimentary name server, which provides string to `GID` mappings, was developed to assist in the testing of the object storage system. Ideally, Raven should support a well-known persistent object accessible from the Raven System object which programmers can use to provide their own name service.

CHAPTER 5

Discussion and Future Work

The design and development of the Raven property scheme is extensive but far from complete. In particular the implementation has brought to light many issues and possible directions for future research.

5.1 Property Scheme Design

The Raven system has experienced only limited use, which has been confined to answering specific research questions. Until Raven is fully implemented and used as the basis for developing distributed applications which rely on all aspects of the property scheme, it will be difficult to conduct a comprehensive evaluation of the Raven property scheme. It's unclear how useful the different properties and features (such as Part-Of) will be for applications programmers, and in particular the usability of the currently unimplemented properties will not be fully known until the system is extensively used. In the testing which has been performed, the system behaves exactly as specified in the property semantics. In particular, transactional processing can be accomplished

by providing objects with the Controlled, Recoverable, and Durable properties, and placing the objects in Dependent relationships.

There is one problem with the current mechanism for providing transactions, however. The Controlled property is designed to prevent concurrent access to an object by multiple *threads*. Now consider the following scenario, as shown in Figure 18. Thread *T*, inside object *A*, performs a Dependent invocation on object *B*. It then performs a non-Dependent invocation on object *C*, which then makes a Dependent invocation on object *B* again. The invocation from *A*→*B* is one transaction, while the invocation from *C*→*B* is another (the invocation from *A*→*C* is not part of the first transaction, since *C* is not Dependent on *A*). The problem arises when the invocation on *C* returns: if a `restore` statement is executed inside *A*, the instance data of *B* will be restored to a previous state as well. However, the state *B* will be restored to is inconsistent with the transaction *C*→*B*, since the state of *C* (which will not be restored) may have relied on the state *B* was in at the time of the transaction.

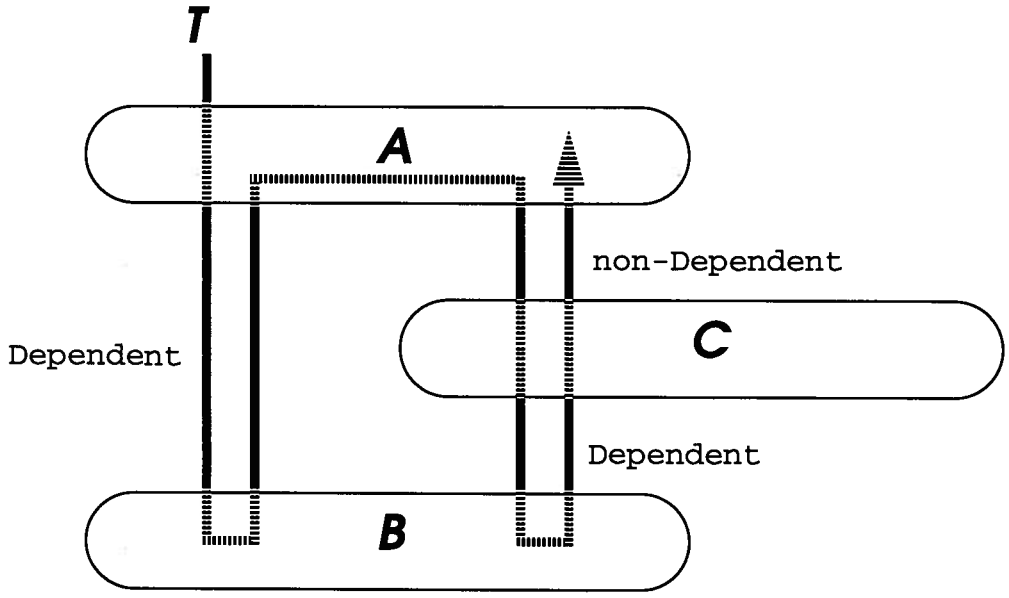

FIGURE 18. Multiple Transactions Modifying a Single Object.

This problem exists because of the particular interaction of the semantics of Dependent and Controlled. The runtime system has a mechanism by which it can differentiate between invocations on an object done by the same thread but as part of different Dependent invocation chains, by using the Dependent `id` to restrict access to the object. However, this does not solve the problem of the semantics, which allow this situation to occur. A solution is to introduce the semantic notion of a invocation identifier (or transaction identifier) which is assigned to every invocation. Invocations in a Dependent chain would all have the same invocation identifier. The semantics of the Controlled property would then be modified to control access to an object based upon the invocation identifier, and not the thread. This is also only a potential problem when the object *B* has the Recoverable property, but due to the orthogonality of the properties the system cannot use its knowledge of the properties of *B* to allow the above situation to occur when *B* is not Recoverable.

When Raven becomes a platform for developing distributed applications, the property scheme may need to be expanded with the addition of new features and capabilities. Potential future features include:

- Property masking: The ability to "mask out" certain properties; that is, a mechanism by which an instance can refuse to be assigned a property which was specified at creation time or inherited from another object.
- Selective inheritance: The ability to selectively bequeath a set of properties at runtime to an instance. Currently, property inheritance requires that all properties be inherited. An extension of this feature would allow *any* set of properties to be dynamically assigned to an object, by specifying the properties when the instance variable is declared in the class definition.
- Alternate policies for system properties: The ability for the user to choose from among a set of different versions of the system properties, which each provide different semantics. For example, the current semantics for Replicated only provide weak consistency. The system could instead provide both weakly and strongly consistent Replication, and allow the user to choose either policy for

an object. Other possibilities include different storage policies for the Persistent property, each of which trade off between more immediate updates to disk and providing more updates in RAM.

- User specified property ordering: as mentioned in Section 3.2, the correct behavior of the system relies upon a careful ordering of the property pre- and post-handlers. Because of this, it is not possible for the user to create their own versions of certain system properties, such as Controlled, since the system must acquire locks before any other work is done and release them only after all other work has completed. To allow users to write their own versions of Controlled, the system could allow the user to specify the ordering of the pre- and post-handlers, thereby installing user written pre- and post-methods as the first and last handlers to be executed. A alternative to this scheme is to continue to hide the actual ordering from the user, but allow the user to load their own property handlers in place of the existing system handlers. Such a scheme would integrate well with providing alternate policies for the properties: the user could choose from among various system supplied properties, or choose one of their own, to install as the Controlled handler, the Persistent handler, etc.

5.1.1 Note on Replication

When the Replicated property is implemented, it will become necessary to decide how Replicated objects should be stored. Should Persistent or Durable Replicated objects be kept on the local disk of all machines where they are Replicated, or perhaps only a subset of these machines? For correctness, the system only needs to keep one non-volatile version of the object, and requiring every node to store the replicas wastes disk space and adds to the system overhead. For a read-mostly database system consisting of one master any many clones, storage is only required (and perhaps desired) on the master node. There may also be situations where "diskless Ravenstations" are used (machines whose only disk is used for swap space, as exist in current Unix systems), in which case it is not possible to do any local storage. To provide such an implementation, however, violates the orthogonality of the properties, which requires that each of the replicas behave as if it has all the properties of the original object (including Persistent or Durable, in which case the

object must be stored to local disk). Even if the user is allowed to select different policies for the behavior of the Persistent or Durable properties, each replica should use the same policy; the object would not be truly replicated if each replica could use different policies.

One option is to provide a new service, similar to Replication, which keeps the instance data of a list of objects in synchronization with each other. As changes were made to one object, they would be propagated to the others. This would allow an object to have many "replicas" each of which could have different properties or policies for their properties.

5.2 Implementation of Property Support

The Raven runtime system is written primarily in C. This lack of reification has actually made parts of the implementation somewhat difficult to accomplish: if an object has the Recoverable property, the system must be able to restore the object's capability information, which is kept in the capability structure and not as part of the instance data. This complicates the recovery process, as the shadow copies used by Recoverable must be augmented with a mechanism to recompute (or save and restore) the capability structure entries. There is also no technical reason why concurrency control locks and shadow copies cannot be implemented using Raven objects, except that such an implementation would result in a drop in performance, as the overhead for performing an invocation is greater than that of a normal C invocation (see [1]).

A possible future implementation is to move the capability information into the instance data. This would provide several benefits:

(1) The existing support for Recoverability could be used to provide recovery support when assigning objects to instance variables which are made `partof` or `inherits`.

(2) It would allow the user access to the information which is currently stored in the capability structure, which is currently unavailable for the user's inspec-

> tion. Raven would therefore be more reflexive: the user could modify this information, changing the object's behavior accordingly. For example, the user could select from among different objects to manage an instance's locking and storage, depending on the behavior the user wishes the object to have, and set the appropriate instance variables to reference these objects. Property behavior could therefore be further customized on an instance by instance basis.

User defined properties are a particularly valuable tool, and present a future direction for the development of all Raven properties: implementing all of the current system properties as user properties. This would force reification of the system properties, since they could no longer rely on C-level data structures, although C data types would still be required to interface with TDBM.

Since user properties are supported through the use of special pre- and post-methods, a reification of the Raven system properties would require that all properties be supported with pre- and post-methods. As with user properties currently, these methods would need to be invoked in a special manner to ensure that they are not invoked recursively when the invocations on the handler methods are made.

The current implementation of user properties does suffer from a significant problem. Because the pre-methods for the user properties are executed after the Recovery pre-handler has executed, if a `restore` statement is executed then the instance data used by the user properties will be reset to the state it was in prior to the method invocation. However, the user post-methods will still be executed, and they may rely on information set by the pre-methods. The problem is further complicated by the mechanics of the `restore` statement: the invocation is not terminated, it instead continues on inside the method after the instance data has been restored. A solution to this problem is to force the system to return control to the caller when a `restore` statement is executed. If all the Raven properties were implemented using Raven objects, the complete property state of the object could then be restored and the post-handlers would not need

to be executed—it would be as if the invocation never took place. (An exception to this scheme would be for the Controlled handlers: locks must always be acquired first, and if any were acquired, they would need to be freed; as such, the Controlled post-handler would still need to be executed.)

The current implementation is also deficient in that it does not completely obey the specified semantics for Dependent invocations. As described in Section 3.7, when a `leave` statement is executed the system will temporarily release (suspend) any locks held on the current object, in case the `result` will be provided by a thread executing a method within the same object. Correct execution in accordance with the semantics of Dependency requires that any locks remain in place until the top level invocation completes. Three possible alternate implementations therefore present themselves:

(1) The system does not release any locks when a `leave` is executed and the object is in a Dependent chain. This would require the programmer to ensure that the `result` statement is executed from a separate object.

(2) The system refuses to assign an object to an instance variable marked `dependent` when one of the object's methods contains a `leave`. A runtime error would instead be generated.

(3) The system suspends the current locks, but does not allow any invocation on the object to fully complete until the suspended thread is resumed. A second thread which accessed the object while the original thread was suspended would itself be suspended until the original thread completed the top level invocation of its Dependent chain. If the original thread did not execute a `restore` statement, the second thread would be resumed normally. If however the original thread executed a `restore`, the second thread would need to be aborted in some manner. The exact semantics and implementation of this scheme would need to be carefully worked out to ensure that orthogonality between the Recoverable and Controlled properties was maintained.

Of these options, number (2) is the most consistent with the current state of the implementation. Since Dependent chains (and therefore transactions) are restricted to a single thread, the system should not allow a second thread to essentially become part of the transaction by providing a return value for an invocation.

Perhaps the most debatable implementation detail is the use of cluster locks for Part-Of clusters. Using cluster locks allows the runtime system to make assumptions about the states of the objects in a cluster, but impacts the user by increasing the overhead associated with every implementation on an object in a cluster. Cluster locks are useful to enforce the consistency of a cluster when the cluster is to be migrated or stored to disk, but are not required for these purposes (e.g., keeping cached versions of the objects in a cluster allows consistent object states to be written to disk). If cluster locks are to remain a part of the implementation, future enhancements to the runtime system should focus on improving the performance of invocations on Part-Of clusters by keeping the amount of locking performed to a minimum. This can be accomplished by bypassing any local concurrency control lock for objects in a cluster, and bypassing the local and cluster locks when one object in the cluster invokes a method on another object in the cluster.

5.3 Object Storage

The object storage system provides reliable storage of objects to disk. Since TDBM was designed to provide efficient atomic storage, supplanting TDBM with a Raven-specific atomic storage scheme is unnecessary. There is currently no mechanism to perform garbage collection on the object store; however the system does provide an "`ls`"-like function to list the `GID`s and class names of all the objects in the store as a potential aid to a human user in cleaning out persistent garbage. A mechanism for collecting distributed, persistent garbage created by Raven applications needs to be developed.

To allow all the objects in a cluster to be loaded from disk into memory together, there is a close relationship between Part-Of clusters and the object storage scheme. This is accomplished by storing all the objects in the cluster together. However, some pieces of a Part-Of cluster may be rather large, in which case the system will perform several large memory-to-memory copies and disk transfers which could be unnecessary if the large object wasn't the piece which was modified. Storing the objects together facilitates the retrieval of the entire cluster from disk, but it is not required. An alternate implementation is for the system to store each of the objects in a Part-Of cluster separately on disk, but load all of the objects when any one object is fetched. This scheme has a further advantage in that it allows the user thread to begin execution inside the first fetched object, while the system fetches the remaining objects of the Part-Of cluster in parallel.

In the current implementation of the storage model, an object's Storage Manager is responsible for creating the full encoding of the object (adding the object's capability information to the swizzled instance data) during object storage, while the GIDmanager extracts the capability information when the object is loaded in from disk. This requires that the GIDmanager and the Storage Managers agree on the general format of the encoded data, so that the GIDmanager can extract the information encoded by a Storage Manager. This is a potential liability, since any change in the encoding format must be reflected in two class implementations. An alternate implementation is for the Storage Managers to pass the swizzled data to the GIDmanager for encoding instead of performing the encoding themselves. This problem is also addressed if the system is modified so that the capability information is moved into the instance data, as the process of swizzling will produce the full object encoding for extraction by the unswizzle routines.

The implementation of Persistent property improves performance by performing the storage in parallel with the execution of the user thread. Performance could be improved significantly if the system could delay the parallel writes, allowing the object state to be updated many times in memory before actually performing the storage. This could be accomplished by having an active

thread inside of every Storage Manager. When an invocation on a storable object finished, the cached version of the object would be updated in the Storage Manager and a boolean flag would be set. The active thread would sleep a specified period of time, then awaken and check the flag to see if the caches were modified, and write them to disk if they were. It would then go back to sleep for its time-out period.

In the current implementation, the system makes a distinction between the case when a single object is stored or a cluster of objects are stored together: in the first case, the encoded data is passed around in a single TDBMDatum object; in the latter, an array of TDBMDatum objects is used. The implementation could be simplified if the system were changed to always use an array of TDBMDatum objects when passing the encoded object data to and from the TDBMmanager. If only one object is to be stored, the array would simply contain one TDBMDatum object. The format of object storage on disk could also be simplified by eliminating the format used by the special case of only a single object being stored to disk (see Figure 16).

5.4 The Raven Collection Classes

Problems with the storage of `Collection` class objects were discussed in Section 4.5. Once a mechanism for storing all the Collection classes is standardized (such as providing specialized swizzle and unswizzle methods for each class) one other issue becomes apparent: the `Collection` classes are not designed to allow the items actually in the collection to be considered Part-Of the `Collection` class object. When items are placed in a Set object, for example, there is no mechanism by which they can be described as being Part-Of the Set. This means that if the Set itself is Part-Of some other object, the items in the Set will not be part of the Part-Of cluster. Similarly, there is no mechanism to specify that the items in a collection should inherit properties or be in a Dependent relationship. Property inheritance, Dependency, and Part-Of clustering stop at the `Collection` class objects. One obvious example of where it would be useful to allow

objects inside a collection to inherit properties is the implementation of a Name Server. The Name Server can be implemented using a Persistent Set. If the items in the Set could be Part-Of the Set, string names can be placed in the Set and become Persistent for the duration of the time they are entries in the Name Server.

This issue can be addressed in one of several ways. The system can provide multiple versions of each of the `Collection` classes. For example, there can be a regular `Set` class, a `SetPartOf` class, a `SetInherits` class, and a `SetDependent` class. These classes would differ only in the how objects inside the collection would be treated. A second solution is to provide an extra parameter to the `constructor` methods of the `Collection` classes. This parameter would mirror the compiler keywords used to specify Part-Of, Inherits, and Dependent, and would indicate how the items in the collection should be viewed:

- `partof`: The items in the collection should be Part-Of the collection.
- `inherits`: The items in the collection should inherit the properties of the `Collection` class object.
- `dependent`: The items in the collection should be Dependent upon the `Collection` class object.

A third solution is for the system to treat the objects inside a collection in the same way the `Collection` class instance is used. If the collection object is Part-Of another object, the objects in the collection will be Part-Of the collection. If the collection object is invoked upon as part of a Dependent chain, the objects in the collection will become part of the same chain. If the collection object inherits properties, then the objects in the collection will inherit properties as well.

CHAPTER 6

Related Work

6.1 Object-oriented Systems

The problem of providing system services in object-oriented systems is not new, and many solutions have been implemented. One scheme is to map traditional services into object representations. These system objects are crafted to integrate into the object model provided by their runtime systems. Another scheme is to provide direct runtime support for various services to objects in the system. There are many object-oriented languages which simply give the user objects, without any real runtime system support. In such systems the user relies on the underlying system (Unix, etc.) for services as they would in a non-object-oriented language. Sometimes objects can be exploited to make access to some services, such as persistence, more automatic.

6.1.1 C++

Several different approaches have been proposed for adding persistence to C++ objects [12], including some which even allow the incremental loading of object data [22]. This has the benefit

of allowing C++ objects existing in any operating system to be persistent, but the consequence is that no general operating system support can be assumed. These approaches rely on overloading the ">>" and "<<" operators to write the object data into streams to which files have been attached. The basic mechanism is for a persistent class to write out a typed stream which includes a class identifier followed by the instance data. These schemes do not provide general persistence for all objects, but provide mechanisms which programmers can use to make specific classes persistent. Since C++ classes aren't real objects, there is no mechanism by which the system can automatically traverse the instance data. Instead, the programmer is responsible for writing the load and store functions for each class by hand (although tools for generating these functions automatically could potentially be developed at an unknown cost). These schemes further require the user to manage a unique identifier (either a class name or numerical id) for each class which is to be persistent, and to develop a mechanism (such as a simple switch or jump table) to allow the creation of new classes given the name or identity.

6.1.2 Arjuna

The Arjuna system ([18], [23]) was designed to provide a C++ class library and system to support the building of robust distributed systems through the use of atomic transactions and persistent objects. Arjuna supplies the system support for atomic transactions (serialized concurrency control and recoverability) and persistent objects through specific C++ classes organized into a class or type hierarchy. Arjuna is currently designed to run on top of Unix. Unlike Raven, multiple versions of what are essentially the same class are required if one version is to make use of the system services for atomic transactions and persistence. Additionally the support for these services is not transparent to the programmer like it is in Raven. The use of transactions, for example, requires the programmer to explicitly instantiate an atomic transaction object and then use methods of the atomic transaction object to start, end or abort the transaction within the object the transaction is to affect. Because services are provided through methods inherited from system

classes, the programmer does have the opportunity to subclass these system supplied classes in order to customize the way they system handles persistence, recoverability, and concurrency control.

6.1.3 NEXTSTEP

Perhaps the most widely used object-oriented system is NEXTSTEP [17], which runs on hardware developed by NeXT Inc. as well as Intel 486 and Pentium platforms. NEXTSTEP is not a true object-oriented operating system, but provides a runtime system for Objective-C objects which sits on top of Mach and Unix. Programmers are required to manage the use of system services themselves. The programmer can make use of the traditional Unix system calls, but the recommended method for managing object archiving is through the use of special C functions provided in NEXTSTEP. These functions support the archiving and retrieval of objects to specially typed streams by writing to the stream the object class and it's instance data. These functions can ensure that objects are only encoded once inside of each stream. By using these functions, the programmer can read and write objects and object references without having to write special functions for each class. Concurrent programming is possible in NEXTSTEP using Mach threads or the cthreads package. No system support for concurrency control is provided, however. It is the programmers responsibility to create mutex locks and manage their use among any cooperating threads.

6.1.4 Argus

The Argus system [13] was designed specifically for use for developing applications that require stable data storage in a distributed environment (such as banking systems). Argus provides special objects called *guardians* which effectively encapsulate data with a well defined interface, and can also shield the data from the effects of network and host failures. Special atomic objects, which contain locks to control against concurrent accesses, are used to synchronize

accesses to guardians. Persistence is achieved by explicitly declaring part (or all) of the data kept by a guardian to be stable when the guardian is designed. Once the programmer has specified atomic and stable objects, the Argus system itself provides serializability and persistence without the programmer having to explicitly make calls to acquire locks or write the data to storage. Other types of system services are not available, as the designers of Argus focused primarily on providing persistent, distributed atomic transactions, and developed special language and runtime system features to support such transactions.

6.1.5 Cronus

Cronus [5] is an object-oriented distributed computing environment developed by BBN Laboratories Inc., that is a capable of connecting groups of heterogenous computers. Among its goals are providing typical operating system services to a distributed environment, simplifying the task of writing distributed programs, building highly available systems, building scalable systems and integrating local and remote resources. To accomplish these goals Cronus employs the active object model with access to objects managed by an object manager. Each machine has one object manager per object type and the object performs pre and post method processing and determines the method to execute. Although there are some routines that can be used to control the execution of an object manager, these techniques apply to the whole collection of objects and not to individual objects. Raven provides a similar type of system control only it is on a per object basis. Cronus does provide a mechanism to allow individual objects to be customized. Each object has methods to allow user specific data to be attached to an object. The interpretation and use of this data, however, is the responsibility of the application using the object. It is essentially advisory information attached to the object that only has a meaning if the user of the object chooses to examine the data and follow any usage guidelines associated with the data.

6.1.6 Guide

Like Raven, Guide [4] consists of both a programming language and a runtime system. Guide is designed for a multi-threaded environment with persistent synchronized objects which support atomic transactions in a distributed environment. In contrast with Raven, all objects in Guide are persistent. Guide also has support for an atomic property for objects, but Guide requires all updates to atomic objects to be within the confines of a transaction. Furthermore, atomicity must be specified at object creation time, and cannot be changed afterwards. Invocations in Guide are accomplished by using the ObjectCall system primitive. This does allow the system to catch invocations on objects, and potentially insert additional code before and after the method execution to provide services in a manner similar to Raven.

6.1.7 Spring

Spring [15] is a new object-oriented distributed operating system developed by Sun Microsystems, Inc. It uses objects to provide strong interfaces to data and services. Although one of the primary motivations for Spring objects is the transparency they provide during distributed computations, Spring objects can also be used within a local address space. One of the key components of a Spring object is its subcontract [9], which defines the communications mechanisms the object should use at runtime. Different subcontracts can be created to provide different services. For example, the architects of Spring have developed subcontracts to provide object replication, caching, and crash recovery, among others. Subcontracts provide a powerful mechanism for customizing the way invocations are made on objects. However, they have some limitations as the basis for a general service providing scheme like Raven properties. First, subcontracts are not generally used for local (same address space) invocations, as Spring's Interface Definition Language (IDL) compiler will transform method invocations into calls on the object's regular method table, bypassing the subcontract, when it can access the object directly. Second, subcontracts cannot be assigned to an object dynamically, to allow the object to change the services it receives; a particu-

lar subcontract is used when the object is created and is thereafter used by the object. This also can lead to a "subcontract explosion" similar to the class explosion problem, as an object could not have both the replicated and crash recoverable subcontracts, for example, but would need some subcontract that combined both services if both were desired. An intriguing notion would be to devise some mechanism to expand the abilities of subcontracts so that they could be composed together and dynamically changed at runtime ala Raven properties.

6.1.8 COOL

COOL [8] is a distributed object-oriented layer developed for the Chorus system [21]. Object descriptors in COOL include an attributes field, which is similar to the properties field of the Raven capability structure. COOL attributes can specify various properties of the object, such as whether the object is persistent or has an active thread. If an object is persistent, then the entire context (address space) in which the object resides is also persistent, and the system will save objects and threads in the context at shutdown time (and checkpoint them during execution) for automatic restart when the system reboots. COOL objects must have a special attribute to be globally known, otherwise they are accessible only from within their context. One main difference between COOL attributes and Raven properties is that attributes are assigned only at object creation time.

6.2 Summary

Raven differs significantly from many of the languages and systems discussed in this section in that it avoids a class explosion when classes differ only in the underlying system support that they require. For example, an object of an existing class can be made persistent, recoverable and controlled simply by instantiating a new instance of that class with the desired properties instead of programming a new class.

Unlike several of the systems described in this section, Raven's support for system services is largely transparent to the programmer and requires little effort on the programmer's part to utilize. Because services are provided automatically by the system, the programmer does not have to manage their use. Raven avoids some of the performance problems inherent in other systems which provide services transparently, because Raven allows objects to be assigned properties on a per-instance basis. Objects only incur performance overhead for services they actually use.

CHAPTER 7

Conclusion

This thesis has presented a new technique for providing system services to objects in an object-oriented system. This technique associates each system service with a property, which when assigned to an object will allow the object to receive the associated service. It also allows the programmer to describe inter-object relationships, to provide object clustering and serialize invocations on objects.

The design and implementation of the property scheme for Raven has shown that it is possible for an object-oriented system to provide services to objects:

- transparently, without the need for the programmer to manage the use of the services, and
- on a per-instance basis, so invocation performance is directly related to the number of services the object receives.

The property scheme also provides atomicity of method invocations through the use of Dependent relationships.

Three aspects of the property scheme stand out above the others: Inherits references, Dependent references, and user properties. These features not only contribute to the uniqueness of the Raven property scheme, they are enabling features which vastly extend the flexibility and usability of the system, providing basic capabilities which can be built upon to achieve desired results.

Bibliography

[1] Donald Acton. Unified Language and Operating Systems Support for Parallel Processing. Ph.D. Thesis PHD094-ACTO, Department of Computer Science, University of British Columbia, Vancouver, British Columbia, 1994.

[2] Donald Acton, Terry Coatta, and Gerald Neufeld. *The Raven System.* Technical Report TR92-15, Department of Computer Science, University of British Columbia, Vancouver, British Columbia, 1992.

[3] Donald Acton and Gerald Neufeld. *Controlling Concurrent Access to Objects in the Raven System.* In 1992 International Workshop on Object-Orientation in Operating Systems, IWOOOS '92. IEEE Computer Society Technical Committee on Operating Systems, September 24-25 1992.

[4] R. Balter et al. *Architecture and Implementation of Guide, an Object-Oriented Distributed System.* In Computing Systems, 4, 1991.

[5] James C. Berets, Natasha Cherniak, and Richard M. Sands. *Introduction to Cronus.* Technical Report 6986, BBN Systems and Technologies, Cambridge, MA, January 1993.

[6] Barry Brachman and Gerald Neufeld. *TDBM: A DBM Library with Atomic Transactions.* Proceedings of the USENIX Summer Technical Conference, June, 1992, pp. 63-80.

[7] Terry Coatta. *Configuration Management Using Objects and Constraints.* PhD Thesis PHD094-COAT, Department of Computer Science, University of British Columbia, Vancouver, British Columbia, 1994.

[8] Sabine Habert and Laurence Mosseri. COOL: Kernel Support for Object-Oriented Environments. Proceedings of ECOOP/OOPSLA 1990, October 21-25 1990, pp.269-277.

[9] Graham Hamilton, Michael L. Powell, and James G. Mitchell. Subcontract: A Flexible Base for Distributed Programming. Proceedings of the 14th Symposium on Operating Systems Principles, Asheville, NC, December 1993.

[10] Norm Hutchinson. *Generalized Table Routines* (unpublished).

[11] The Institute of Electrical and Electronic Engineers. *Information Technology—Portable Operating System Interface (POSIX) Part 1: System Application Program Interface.* IEEE Standard 1003.1-1990 (also available as International Standard ISO/IEC 9945-1, 1990).

[12] Philippe Laurent and Nino Silverio. *Persistence in C++.* Journal of Object-Oriented Programming, Vol. 6, No. 6, October 1993, pp. 41-46.

[13] Barbara Liskov. *Distributed Programming In Argus.* Communications of the ACM, Vol. 32, No. 3, March 1988, pp. 300-312.

[14] Mary E.S. Loomis. *The ODMG Object Model.* Journal of Object-Oriented Programming, Vol. 6, No. 3, June 1993, pp.64-69.

[15] James G. Mitchell et al. An Overview of the Spring System. Proceedings of Compcon Spring 1994, February 1994.

[16] Gerald W. Neufeld, Murry W. Goldberg, and Barry J. Brachman. *The UBC OSI Distributed Application Programming Environment.* Technical Report 90-37, Department of Computer Science, University of British Columbia, Vancouver, British Columbia, January 1991.

[17] NeXT, Inc. *NeXTStep Reference.* Addison-Wesley Publishing Company, 1991.

[18] Graham D. Parrington. *Reliable Distributed Programming in C++: The Arjuna Approach.* Technical report, COmputing Laboratory, University of Newcastle upon Tyne, UK.

[19] Stuart Ritchie. *The Raven Kernel: a Microkernel for Shared Memory Multiprocessors.* Masters' Thesis, Department of Computer Science, University of British Columbia, Vancouver, British Columbia, April 1993.

[20] Mendel Rosenblum and John Ousterhout. *The Design and Implementation of a Log-Structured File System.* In Proc. ACM Symposium on Operating Systems Principles, pages 1-15, October 1991.

[21] M. Rozier et al. *Chorus Distributed Operating Systems.* Computing Systems, 1(4):305-367, 1988.

[22] John J. Shilling. *How to Roll Your Own Persistent Objects in C++.* Journal of Object-Oriented Programming, Vol. 7, No. 4, July-August 1994, pp. 25-32.

[23] Santosh K. Shrivastava, Graeme N. Dixon, and Graham D. Parrington. *An Overview of the Arjuna Distributed Programming System.* Technical report, Computing Laboratory, University of Newcastle upon Tyne, UK.

[24] Soley, R.M., Ed. *Object Management Architecture Guide, Rev. 2.0, 2nd Ed.* OMG TC Document 92.11.1, Object Management Group, 1992.

[25] Sun Microsystems Inc.. *XDR: External Data Representation Standard (RFC 1014).* Network Information Center, SRI International, June 1987.

Appendix: Table of Contents

Appendix A Data Types

The implementations of the property scheme and of object storage required modifications to existing data types used by the Raven runtime as well as the development of new data structures. What follows is an exhaustive list of all modified and new data structures used by the runtime, including those already presented in the thesis.

Many of these data types and structures rely upon definitions of other data types and structures. This additional definitions are generally not provided here.

A.1 Capability Structure

This structure is also described in Figure 6.

```
struct capability
{
    funcptr             invoke;
```

`invoke` is a pointer to the current invocation function to use.

```
cap                 id;
```

`id` points to the current capability structure. It is useful when debugging as a sanity check for a capability pointer.

```
cap                 is_a;
```

`is_a` points to the Class object which this object is an instance of.

```
cap                 parent;
```

`parent` points to the object from which this object inherits properties. parent points to the `nil` object if we do not inherit properties.

```
cap                 inh_root;
```

`inh_root` points to the object which lies at the top of our current inheritence chain. This pointer points to ourself if we do not inherit properties. This allows us to detect inheritance cycles, such as attempting to inherit properties from ourself.

```
cap                 storage_manager;
```

`storage_manager` points to our current Storage Manager object, or to `nil` if we do not have the Persistent or Durable properties.

```
method_type         method_type_to_use;
```

`method_type_to_use` is used by the runtime system to determine if a local version of a method exists. Some invocations on remote objects can (and should) be handled locally.

```
struct gid          *gid;
```

`gid` points to a global identification structure (see Section A.6). In general this will point to NULL unless the object requires such a structure.

```
voidp               rw_lock;
```

`rw_lock` is a pointer to the object's concurrency control lock, or to NULL if the object is not Controlled. The structure of this lock is shown in Section A.4.

```
voidp               cluster_lock;
```

`cluster_lock` is a pointer to the object's cluster lock, or to NULL if the object is not part of a Part-Of cluster.

```
properties          object_properties;
```

`object_properties` contains the current property set of the object.

```
u_char              *data;
```

data is a pointer to the instance data for the object.

```
} ;
```

A.2 Property Values

```
typedef u_long        properties;
```

A.2.1 Property Masks

```
#define default_size                0x10
#define inherited_mask              0xffff0000
#define default_mask                0x0000ffff
#define no_properties               0xffffffff
#define basic                       no_properties
#define persistent                  0xfffffffe
#define inherited_persistent        0xfffeffff
#define durable                     0xfffffffd
#define inherited_durable           0xfffdffff
#define recoverable                 0xfffffffb
#define inherited_recoverable       0xfffbffff
#define controlled                  0xfffffff7
#define inherited_controlled        0xfff7ffff
#define immutable                   0xffffffef
#define inherited_immutable         0xffefffff
#define immobile                    0xffffffdf
#define inherited_immobile          0xffdfffff
#define replicated                  0xffffffbf
#define inherited_replicated        0xffbfffff
#define test_prop                   0xffffff7f
#define inherited_test_prop         0xff7fffff
#define u_prop_1                    0xfffffeff
#define inherited_u_prop_1          0xfeffffff
```

```
#define u_prop_2                    0xfffffdff
#define inherited_u_prop_2          0xfdffffff
#define u_prop_3                    0xfffffbff
#define inherited_u_prop_3          0xfbffffff
#define u_prop_4                    0xfffff7ff
#define inherited_u_prop_4          0xf7ffffff
```

A.2.2 System Supported Properties

```
typedef enum
{
    LOCKING = 0,
    PERSISTENCE,
    DURABLE,
    RECOVERABILITY,
    IMMUTABILITY,
    REPLICATION,
    IMMOBILITY,
    NUM_PROPERTIES
} property_categories;
```

A.3 Structures for Supporting the Recoverable Property

Recoverability is accomplished by creating shadow copies of the object instance data prior to executing the method code. The format of the shadow structure has been modified to support Dependency.

A.3.1 Shadow Structure

```
typedef struct ShadowStruct
{
    struct ShadowStruct*next;
```

next points to the next Shadow structure in the chained list of shadows.

```
    struct ShadowStruct *previous;
```

previous points to the previous Shadow structure in the chained list of shadows.

```
    struct capability *object;
```

object is a pointer to the object for which this is a Shadow structure.

```
    u_long              methodHash;
```

methodHash is used for debugging purposes when dumping out the shadow list.

```
    int                 depth;
```

depth is the current call depth in the shadow chain.

```
    char                *image;
```

image is a pointer to the copy of the instance data of the object.

```
    int                 invokeDepth;
```

invokeDepth is incremented once for each invocation made on a Recoverable object in the current Dependent calling chain.

```
    int                 dependentID;
```

dependentID is the ID of the current Dependent calling chain.

```
} Shadow;
```

A.4 Structures for Supporting the Controlled Property

Each controlled object has an associated concurrency control lock. Objects which are in a Part-Of cluster have a seperate cluster lock. Each thread maintains a list of the locks which it has aquired.

A.4.1 Lock Status Values

Each lock which a thread holds (or is attempting to aquire) has an associated status.

```
typedef enum
{
    GRANTED = 1,
```

Indicates the lock has been granted to the thread.

`WAITING,`

Indicates the thread is waiting to aquire the lock.

`SUSPENDED,`

Indicates the thread has been suspended and the lock temporarily relinquished by the thread due to a delayed result.

`WAITING_REACQUIRE,`

Indicates the thread is waiting to reaquire a lock that had been suspended.

`RETRY,`

Indicates that the thread should attempt to aquire the lock again. This will generally involve the thread reexecuting the locking portions of the property pre-handlers.

`DELETED,`

Indicates that the specified lock was deleted, either becuase the object no longer is Controlled or is no longer in a Part-Of cluster.

`LOCKING_ERROR,`

Not currently used in the implementation.

`LOCK_TIMED_OUT`

Indicates the lock could not be granted before the timeout timer expired. A possible indication of deadlock.

```
} LOCK_STATUS;
```

A.4.2 Lock Types

```
typedef enum
{
    CLUSTER_LOCK = 1,
    INSTANCE_LOCK
} LOCK_FAMILIES;
```

A.4.3 Lock Structure

The concurrency control and cluster locks have the following structure.

```
typedef struct LOCK
{
    LIST                granted;
```

The list of threads which have been granted the lock.

```
    LIST                waiting;
```

The list of threads which are waiting for the lock.

```
    LIST                suspended;
```

The list of threads which have temporarily released this lock due to a delayed result.

```
    LIST                reacquire_list;
```

The list of threads which had temporarily released the lock and now wish to aquire it again. They are given preferrential access to the lock over threads which are simply on the `waiting` list.

```
    OSSemaphore         semaphore;
```

The semaphore which ensures that only one thread is manipulating the lock data structures at a time.

```
} LOCK;
```

A.4.4 Lock Information Structure

Each entry on any of the lock lists is a `LOCK_INFO` structure. Threads maintain a chain of these structures, one structure for each lock they have aquired, are waiting for, or are suspended in.

```
typedef struct LOCK_INFO
{
    struct LOCK_INFO  *next;
```

Used to point to the next structure in the list (granted, waiting, etc.) we are currently on.

```
    struct LOCK_INFO  *previous;
```

Used to point to the previous structure in the list (granted, waiting, etc.) we are currently on.

```
    struct LOCK_INFO  *session_chain_ptr;
```

The next structure in the chain of LOCK_INFO structures held by the thread.

```
struct LOCK        *lock_ptr;
```

A pointer to the lock.

```
void               *waiting_thread;
```

If the thread is currently suspended, it's position is recorded here.

```
behaviorLockType  lock_type;
```

The type of lock (read, write) we are holding/wish to hold.

```
int                lock_holder;
```

The PID of the process.

```
cap                objectID;
```

The object which was was invoked upon.

```
int                session_id;
```

The identifier of the current session (unique for each thread).

```
int                lock_depth;
```

The current depth in the locking chain.

```
LOCK_STATUS        lock_status;
```

The status of our attempt to aquire a lock will be placed here.

```
int                dependentID;
```

The id of the current dependent chain.

```
} LOCK_INFO;
```

A.5 Structures for Supporting Object Storage

Each thread maintains a list of Storage Managers for the objects that have been invoked upon in a Dependent calling chain. The entries of this list are of type `struct STORAGE_INFO`.

A.5.1 Storage Information Structure

```
typedef struct STORAGE_INFO
{
```

```
    struct STORAGE_INFO *next;
```

The next entry on the list of Storage Managers.

```
    struct STORAGE_INFO *previous;
```

The previous entry on the list of Storage Managers.

```
    int                 dependentID;
```

The identifier of the current Dependent calling chain.

```
    cap                 manager;
```

The Storage Manager object.

```
    LIST                *objects;
```

A list of objects managed by the Storage Manager which may have been modified during the current Dependent invocation chain. Each entry is of type STORELIST_ENTRY, which is shown below.

```
} STORAGE_INFO;
```

A.5.2 Object List Entry Structure

```
typedef struct STORELIST_ENTRY
{
    struct STORELIST_ENTRY *next;
```

The next entry in this list.

```
    struct STORELIST_ENTRY *previous;
```

The previous entry in this list.

```
    cap                 object;
```

The object, managed by the Storage Manager, which was invoked upon.

```
} STORELIST_ENTRY;
```

A.6 Unique Identification Structures

An object's unique identifier is stored in three levels. This is partly a historical artifact, due to thoughts about how Raven would handle gids. Raven's initial use of the term "gid" is somewhat of a misnomer, since it was simply used to describe object location information (which,

prior to object storage, was sufficient to uniquely identify an object). The capability structure of an object contains a pointer to a `struct gid`.

A.6.1 GID Structure

```
struct gid
{
    struct u_gid        u_gid;
```

The structure which actually contains the location information and unique identification information.

```
    int                 gid_len;
```

Unused. Originally designed to hold the length of the encoded `gid`.

```
    char                *encoded_gid;
```

Unused. Originally designed to hold a pointer to an encoded version of the `gid`.

```
};
```

A.6.2 Location Information GID Structure

```
struct u_gid
{
    u_long              hid;
```

The IP address of the host on which this object currently resides.

```
    u_long              lid;
```

The port number of the Raven World on which the object currently resides.

```
    struct unique_id    uid;
```

The unique identifier of the object.

```
    d_cap               capability;
```

A pointer to the capability for the object, i.e., the object's location in memory on the Raven World specified by the `hid` and `lid` fields.

```
    char                *class_name;
```

The class name which this object is an instance of.

```
};
```

A.6.3 Unique Identifier Structure

```
struct unique_id
{
    u_long              creator;
```

The IP address of the machine on which the object was created.

```
    u_long              world;
```

The port number to which the Raven World on which the object was created was bound.

```
    u_long              generation;
```

A counter which is incremented once for each incarnation of a Raven World.

```
    u_long              name;
```

A counter which is incremented once for each `gid` which is assigned by an incarnation of a Raven World.

```
};
```

A.7 Compiler Types

A.7.1 Property Set

```
typedef enum
{
    persistent = 0,
    durable,
    recoverable,
    controlled,
    immutable,
    immobile,
    replicated,
    test_prop,
    u_prop_1,
```

```
    u_prop_2,
    u_prop_3,
    u_prop_4,
    phaseout,
    lastProperty
} Properties;
```

The phaseout property is used to warn the user that he has selected a property which is no longer supported by the system.

A.7.2 Variable Attributes

```
typedef enum
{
    stat=0,
    meta,
    copy,
    partof,
    inherits,
    dependent,
    indirect,
    public,
    lastVarAttribute
} VariableAttributes;
```

A.8 Other Data Types

A.8.1 Instance Variable Modifier Bits

```
#define PT_DEPENDENT            0x8

#define PT_INHERITS             0x10
```

```
#define PT_INDIRECT             0x20

#define PARTOF_MASK_VAL         0x40

#define PT_PARTOF               (PT_DEPENDENT | PT_INHERITS | \
                                 PARTOF_MASK_VAL)
```

A.8.2 Defined Values for Object Storage

```
#define GENERATIONSTRING      ".sysGENERATION"
#define GENERATIONSTRINGLEN   15
#define DBMPATH               "/raven/export.r/generic/storage"
```

A.8.3 Environment Variables

RVSTORAGEPATH	Directory into which TDBM will store objects
RAVENWORLD	Port number to bind to when starting Raven

Appendix B

Class Definitions

Many new classes were created during the implementation of the object property scheme. Additionally, many existing classes were modified to support the new property scheme. These classes are presented here.

Some of the interfaces listed here as private interfaces are private simply in that they are not to be known by the general programmer. They are not necessarily private in that they are functions which are only used from within an instance of that class. Many "private" behaviors are invoked by the runtime system, or from other objects which are used to provide support for properties.

B.1 `Thread` Class

B.1.1 Public Interface

```
class Thread uncontrolled
{
```

```
pid, stack_size : Int;
name, instance: Cap;
method : String;
prio : Int;
session_id, session_chain, lock_depth : Int;
shadows, callDepth : Int;
function_chain : Int;
storage_chain : Int;
remoteResumeStatus : Int;
constructor(pprio:Int);
behaviour starter(...);
behaviour start(...) : Int;
behaviour startAsSystemProc(...) : Int;
behaviour attach(obj:Cap) : Int;
behaviour getPid() : Int;
behaviour threadRunnable() : Int;
behaviour kill() : Int;
behaviour threadName() : String;
behaviour resume() : Int;
behaviour stackSize() : Int;
behaviour suspend();
behavior sleep(duration:Int);
behavior startNetwork();
behavior monitorLocks();
behav traceEnable(minSize: Int);
behav traceDisable();
behav traceStackOnly();
behav traceDumpOnFill();
behav traceDumpOnExit();
behav traceDump();
}
```

B.1.2 Private Interface

```
class Thread
{
 behavior remoteDependentWorker(handler : Int,
                                resumeStatus : Int);
 behavior remoteRestore();
 behavior remoteRestoreFunctions();
}
```

B.2 StorageManager Class

B.2.1 Public Interface

```
class StorageManager controlled
{
$objects : FixedArray[cap];
$size : Int;
$entries : Int;

constructor();
behav updateCacheOfObject(obj : cap, doWrite: Int,
                          now : Int) writelock;
behav manageObject(obj : cap);
behav unmanageObject(obj : cap);
behav writeToStorage(writeNow : Int, id : Int);
}
```

B.2.2 Private Interface

```
class StorageManager
{
behav loadDataFromCache() readlock;
behav setCacheValue(datum : cap) writelock;
```

```
behav privateWrite(id : Int) private;
behav getCap(uidPtr : Int) : cap private;
behav privateDelete(entry : SMEntry) private;
behav setUpEntry(obj : cap, entry : cap) private;
behav entriesHaveSameValue(e1 : cap, e2 : cap) : Int private;
}
```

B.3 SMEntry Class

This class is used only internally by the Storage Manger. Its interface is not visible to any other object.

```
class SMEntry <- Object
{
$dataPtr : Int public;
$dataLen : Int public;
$objCap : cap public;
behav loadData() private;
}
```

B.4 GIDManager Class

B.4.1 Public Interface

```
class GIDManager controlled
{
table : Int;
nameList : cap;
creator : Int;
world : Int;
generation : Int;
name : Int;
```

```
constructor();
behav getGID(obj : cap);
behav fetch(obj : cap) : Int writelock;
behav becomeVolatile(obj : cap) writelock;

behav listNames() readlock;
behav findName(name : String) : cap writelock;
behav addName(name : String, obj : cap) : Int writelock;
behav deleteName(name : String) : Int writelock;
}
```

B.4.2 Private Interface

```
class GIDManager
{
behav capFromGID(uID : Int) : cap writelock;
behav findForRemote(uID : Int) : cap writelock;
}
```

B.5 TDBMManager Class

B.5.1 Public Interface

```
class TDBMManager controlled
{
dbm : Int;
namedbm : Int;
recovery : Int;

constructor();
behav shutdown();
behav getGeneration() : Int writelock;
```

```
behav getTID() : Int readlock;
behav preCommit(id : Int) writelock;
behav commit(id : Int) writelock;
behav store(datum : cap) writelock;
behav storeForTransaction(id : Int, datum : cap) writelock;
behav fetch(datum : cap) : cap readlock;
behav delete(datum : TDBMDatum) : Int writelock;
behav dumpObjects() readlock;
behav storename(datum : TDBMDatum) writelock;
behav fetchname(datum : TDBMDatum) : cap readlock;
behav deletename(datum : TDBMDatum) : Int writelock;
behav dumpnames() readlock;
}
```

B.5.2 Private Interface

```
class TDBMManager
{
behav privateStore(transID : Int, datum : cap) nolock;
}
```

B.6 `TDBMDatum` Class

B.6.1 Public Interface

```
class TDBMDatum
{
keydptr : Int;
keydsize : Int;
valuedptr : Int;
valuedsize : Int;

behav setKey(dptr : Int, dsize : Int);
```

```
behav setValue(dptr : Int, dsize : Int);
behav getKeyPtr() : Int;
behav getValuePtr() : Int;
behav getValueSize() : Int;
}
```

B.6.2 Private Interface

There are no additional methods or instance variables beyond what is specified in the public interface.

B.7 New Basic Methods

The `Object` class was extended to include several new methods to do the following:

- Assign the object a GID.
- Swizzle and unswizzle the instance data.
- Provide support for the user defined properties.

These methods were added to the list of basic methods supported by the `Object` class. As such, they do not appear in the interface definition of `Object`.

```
behavior assignGID();
behavior sWizzleData() : Int;
behavior unsWizzleData(data : Int);
behavior preUser1(dependentInvoke : Int);
behavior postUser1(dependentInvoke : Int, isDependentRoot : Int,
                   hasProp : Int);
behavior preUser2(dependentInvoke : Int);
behavior postUser2(dependentInvoke : Int, isDependentRoot : Int,
                   hasProp : Int);
behavior preUser3(dependentInvoke : Int);
```

```
behavior postUser3(dependentInvoke : Int, isDependentRoot : Int,
                   hasProp : Int);
behavior preUser4(dependentInvoke : Int);
behavior postUser4(dependentInvoke : Int, isDependentRoot : Int,
                   hasProp : Int);
```