Chapter

1 Introduction to GemStone

This chapter introduces you to the GemStone system. GemStone provides a distributed, server-based, multiuser, transactional Smalltalk runtime system, Smalltalk application partitioning technology, access to relational data, and production-quality scalability and availability. The GemStone object server allows you to bring together object-based applications and existing enterprise and business information in a three-tier, distributed client/server environment.

1.1 Overview of the GemStone System

GemStone provides a wide range of services to help you build objects-based information systems. GemStone:

- is a multi-user object server
- is a programmable server object system
- manages a large-scale repository of objects
- supports partitioning of applications between client and server
- supports queries and indexes for large-scale object processing
- supports transactions and concurrency control in the object repository
- supports connections to outside data sources
- provides object security and account management
- provides services to manage the object repository.

Each of these features is described in greater detail in the following sections.

1.2 Multi-User Object Server

GemStone can support over 1,000 concurrent users, object repositories of up to 100 gigabytes, and sustained object transaction rates of over 100 transactions per second. Server processes manage the system, while user sessions support individual user activities. Repository and server processes can be distributed among multiple machines, and shared memory and SMP can be leveraged.

Multiple user sessions can be active at the same time, and each user may have multiple sessions open. A flexible naming scheme allows separate or shared namespaces for individual users. Coherent groups of objects can be distributed through replication. Changes users make to objects are committed in transactions, with concurrency controls and locks ensuring that multi-user changes to objects are coordinated. Security is provided at several levels, from login authorization to object access privileges.

1.3 Programmable Server Object System

GemStone provides data definition, data manipulation, and query facilities in a single, computationally complete language — GemStone Smalltalk. The GemStone Smalltalk language offers built-in data types (classes), operators, and control structures comparable in scope and power to those provided by languages such as C, C++, or Pascal, in addition

to multi-user concurrency and repository management services. All system-level facilities, such as transaction control, user authorization, and so on, are accessible from GemStone Smalltalk.

This manual discusses the use of GemStone Smalltalk for system and application development, particularly those aspects of GemStone Smalltalk that are unique to running in a multi-user, secure, transactional system. See the *GemStone System Administration Guide* for more information about system administration functions.

1.4 Partitioning of Applications Between Client and Server

GemStone applications can access objects and run their methods from a number of languages, including Smalltalk, C, C++, or any language that makes C calls (such as COBOL or Fortran). Objects created from any of these languages are interoperable with objects created from the other languages, and can run their methods within GemStone.

To provide this functionality, GemStone provides interface libraries of Smalltalk classes, C++ classes and functions, and C functions. These language interfaces, known collectively as GemBuilder, allow you to move objects between an application program and the GemStone repository, and to connect client objects to GemStone objects. GemBuilder also provides remote messaging capabilities, client replicates, and synchronization of changes.

GemBuilder for Smalltalk is a set of classes installed in a client Smalltalk image that provides access to objects in the GemStone repository. The client Smalltalk application can use these classes to gain access to all of GemStone's production capabilities. GemBuilder for Smalltalk also supports *transparent* GemStone access from a Smalltalk application — client Smalltalk and GemStone objects are related to each other, and GemBuilder maintains the relationship and propagates changes between these client Smalltalk and GemStone objects, not the application.

GemBuilder for C is a library of C functions that provide a bridge between an application's C code and the GemStone object repository. You can work with GemStone objects by importing them into the C program using structural access or by sending messages to objects in the repository through GemStone Smalltalk. You can also call C routines from within GemStone Smalltalk methods.

GemBuilder for C++ provides both persistent storage for C++ applications and access to persistent GemStone objects from applications written in C++. Because C++ objects stored in GemStone take on identity and exist independently of the program that created them, they can be used by other applications, including those written in other programming languages.

Your GemStone system includes one or more of these interfaces. Separate manuals available for each of the GemBuilder products provide full documentation of the functionality and use of these products.

1.5 Large-Scale Repository

Object programming languages such as Smalltalk have proven to be highly efficient development tools. Smalltalk exploits inheritance and code reuse and provides the flexibility of modeling real world objects with self-contained software modules. Most Smalltalk implementations, however, are memory based. Objects are either not saved between executions, or they are saved in a primitive manner that does not lend itself to concurrent usage or sharing. Smalltalk programmers save their work in an "image," which is a file that stores their development environment on a workstation. The image holds the application's classes and instances, the compiled code for all executable methods, and the values of the variables defined in the product.

GemStone is based on the Smalltalk object model—like a single-user Smalltalk image, it consists of classes, methods, instances and meta objects. Persistence is established by attaching new objects to other persistent objects. All objects are derived from a named root (AllUsers). Objects that have been attached and committed to the repository are visible to all other users. However, unlike client Smalltalks with memory-based images, the GemStone repository is accessed through disk caches, so it is not limited in size by available memory. A GemStone repository can contain over a billion objects. Repositories can be distributed among many different machines and files. Because each object in a repository has a unique object identifier (known as an OOP—object-oriented pointer), GemStone applications can access any object without having to know its physical location.

1.6 Queries and Indexes

GemStone lets you model information in structures as simple as the data permits, and no more complex than the data demands. You can represent data objects in tables, hierarchies, networks, queues, or any other structure that is appropriate. Each of these objects may also be indexable. Complex data structures can be built by nesting objects of various formats.

The power and flexibility of GemStone Smalltalk allow you to perform regular and associative access queries against very large collections. Because you can represent information in forms that mirror the information’s natural structure, the translation of user requests into executable queries can be much easier in GemStone. You do not need to translate users’ keystrokes or menu selections into relational algebra formulas, calculus expressions and procedural statements before the query can be executed. See Chapter 5, "Querying.".

1.7 Transactions and Concurrency Control

Each GemStone session defines and maintains a consistent working environment for its application program, presenting the user with a consistent view of the object repository. The user works in an environment in which only his or her changes to objects are visible. These changes are private to the user until the transaction is committed. The effects of updates to the object repository by other users are minimized or invisible during the transaction. GemStone then checks for consistency with other users' changes before committing the transaction.

GemStone provides two approaches to managing concurrent transactions:

- Using the *optimistic* approach, you read and write objects as if you were the only user, letting GemStone manage conflicts with other sessions only when you try to commit a transaction. This approach is easy to implement in an application, but you run the risk of discarding the work you've done if GemStone detects conflicts and does not permit you to commit your transaction. When GemStone looks for conflicts only at your commit time, your chances of being in conflict with other users increase both with the time between your commits and the number of objects being read and written.
- Using the *pessimistic* approach, you prevent conflicts as early as possible by explicitly requesting locks on objects before you modify them. When an object is locked, other users are unable to lock that object or to commit any changes they have made to the object. When you encounter an object that another user has locked, you can wait, or abort your transaction immediately, instead of wasting time doing work that can't be committed. If there is a lot of competition for shared information in your application, or your application can't tolerate even an occasional inability to commit, using locks may be your best choice.

GemStone is designed to prevent conflicts when two users are modifying the same object at the same time. However, some concurrent operations that modify an object, but in consistent ways, should be allowed to proceed. For example, it might not cause any concern if two users concurrently added objects to the same Bag in a particular application.

For such cases, GemStone provides reduced-conflict (Rc) classes that can be used instead of the regular classes in those applications that might otherwise experience too many unnecessary conflicts:

- *RcCounter* can be used instead of a simple number for keeping track of amounts when it isn't crucial that you know the results right away.
- *RcIdentityBag* provides the same functionality as IdentityBag, except that no conflict occurs if a number of users read objects in the bag or add objects to the bag at the same time.

- *RcQueue* provides a first-in, first-out queue in which no conflict occurs when other users read objects in the queue or add objects to the queue at the same time.
- *RcKeyValueDictionary* provides the same functionality as KeyValueDictionary, except that no conflict occurs when users read values in the dictionary or add keys and values to the dictionary at the same time.

See Chapter 6, "Transactions and Concurrency Control."

1.8 Connections to Outside Data Sources

While GemStone methods are all written in Smalltalk (except for a few primitives), you may often want to call out to other logic written in C. GemStone provides a way to attach external code, called userActions, to a GemStone session. With userActions, you can access or generate external information and bring it into GemStone as objects, which can then be committed and made available to other users. GemBuilder for C is used to write userActions in C and add them to GemStone Smalltalk, according to rules described in the *GemBuilder for C* manual. The description of class System in the *GemStone Kernel Reference* describes the messages you can send to invoke these userActions.

GemStone uses this mechanism to build its GemConnect product, which provides access to relational database information from GemStone objects. GemConnect also provides automatic tracking of object modifications for synchronizing the relational database, and supports the generation of SQL to update the relational database with changes. GemConnect is fully encapsulated and maintained in the GemStone object server. Refer to the *GemConnect Programming Guide* for more information about GemConnect and its capabilities.

1.9 Object Security and Account Management

Compared to a single-user Smalltalk system, GemStone requires substantially more security mechanisms and controls. As a tool for server implementation, multi-user Smalltalk must handle requests from many users running a variety of applications, each of which can require different accessibility of objects. Authentication and authorization are the cornerstones of GemStone Smalltalk security.

A server must reliably identify the people attempting to use a system resource. This identification process is known as authentication. Authentication requires a valid user ID and password. Preventing unauthorized users from entering the system by requiring user names and passwords is generally effective against casual intrusion. GemStone Smalltalk supports its own authentication protocol, as well as the Kerberos scheme.

The next type of security, known as authorization, exists within GemStone and controls individual object access. Authorization enforcement is implemented at the lowest level of basic object access to prevent users from circumventing the authorization checking. No object can be accessed from any language without suitable authorization. GemStone provides a number of classes to define and manage object authorization policies. These classes are discussed in greater detail in this manual.

Finally, GemStone defines a set of **privileges** for controlling the use of certain system services. Privileges determine whether the user is allowed to execute certain system functions usually only performed by the system administrator. Privileges are more powerful than authorization. A privileged user can override authorization protection by sending privileged messages to change the authorization scheme.

In GemStone Smalltalk, a user is represented by an instance of class UserProfile. A UserProfile contains the following information about a user:

- unique userID,
- password (encrypted),
- default authorization information,
- privileges,
- group memberships.

Only users who have a UserProfile can log on to the system. For more information on UserProfile, see the *GemStone System Administration Guide*.

See Chapter 7, "Object Security and Authorization."

1.10 Services to Manage the GemStone Repository

GemStone objects are often an enterprise resource. They must be shared among all users and applications to fill their role as repositories of critical business information and logic. Their role goes beyond individual applications, requiring permanence and availability to all parts of the system. GemStone is capable of managing large numbers of objects shared by hundreds of users, running methods that access millions of objects, and handling queries over large collections of objects by using indexes and query optimization. It can support large-scale deployments on multiple machines in a variety of network configurations. All of this functionality requires a wide array of services for management of the repository, the system processes, and user sessions.

GemStone provides services that can:

- support flexible backup and restore procedures,
- recover from hardware and network failures,
- perform object recovery when needed,
- tune the object server to provide high transaction rates by using shared memory and asynchronous I/O processes,
- accommodate the addition of new machines and processors without recoding the system,
- make controlled changes to the definition of the business and application objects in the system.

This manual provides information about programmatical techniques that can be used to optimize your GemStone environment for system administration. Actual system administration and management processes are discussed in the *GemStone System Administration Guide*.

Chapter

2 Programming With GemStone

This chapter provides an overview of the programming environment provided by GemStone.

The GemStone Programming Model
describes how programming in GemStone differs from programming in a client Smalltalk development environment.

GemStone Smalltalk
explains the unique aspects of GemStone Smalltalk that affect programming and application design.

GemStone Architecture
describes GemStone's development and runtime process architecture, and how that architecture influences your programming design and techniques.

2.1 The GemStone Programming Model

GemStone is an object server, so programming with GemStone is somewhat different than programming with a client Smalltalk development environment. However, there is a great deal that GemStone has in common with client Smalltalk development, so many of the programming concepts will be quite familiar to you if you have previously worked with a client Smalltalk system.

Server-based classes, methods, and objects

One key characteristic of GemStone programming is that GemStone Smalltalk runs in a server, not in a client. Running in a server means that GemStone classes and methods are stored in a server-based repository (image), and activated by processes which run on a server, often without a keyboard or screen present. The developer writing GemStone classes and methods is usually working at a client machine, communicating with the GemStone environment remotely.

Running in a server also means that the services provided by GemStone's own class library are oriented toward server activity. GemStone's class library provides functionality for:

- data handling
- collection processing and query processing
- system management
- user account management

The GemStone class library does not provide functionality for screen presentation and user interface issues. User interface functionality is provided in client Smalltalk products.

Because GemStone is an object server, it provides a large number of mechanisms for communicating with GemStone objects from remote machines for development purposes, application support, and system management. Remote machines often host a programming environment that communicates with GemStone through a GemStone interface. A significant part of programming with GemStone is designing the interactions between various client and server-based runtime systems and the GemStone classes, methods, and objects created by the developer.

Client and Server Interfaces

GemStone provides a number of client and server interfaces to make it easy for developers to write applications which make use of GemStone objects, and to write GemStone classes and methods which make use of external data. While an entire application can be built in GemStone Smalltalk and run in the GemStone server, most applications include either a user interface or interaction of some kind with other systems. In addition, management of a running GemStone system involves using GemStone tools and interfaces to program control activities tailored to specific system environments.

GemStone's interfaces are numerous. They include:

GemBuilder for Smalltalk
GemBuilder for Smalltalk consists of two parts: a set of GemStone programming tools, and a programming interface between the client application code and GemStone. GemBuilder for Smalltalk contains a set of classes installed in a client Smalltalk image that provides access to objects in a GemStone repository. Many of the client Smalltalk kernel classes are mapped to equivalent GemStone classes, and additional class mappings can be created by the application developer.

GemBuilder for C++
GemBuilder for C++ provides both shared storage for C++ applications and access to shared objects stored in GemStone by applications written in other languages. GemBuilder for C++ is implemented as a preprocessor based on standard C++ syntax. A class library is provided, giving the programmer a standard set of definitions for commonly used data structures such as sets, arrays, and bags, as well as functions for managing and manipulating GemStone objects with C++ code.

GemBuilder for C
GemBuilder for C is a library of C functions that provide a bridge between an application's C code and the GemStone repository. This interface allows programmers to work with GemStone objects by importing them into the C program using structural access, or by sending messages to objects in the repository through GemStone Smalltalk. C routines can also be called from within GemStone Smalltalk methods.

Topaz
Topaz is a GemStone programming environment that provides keyboard command access to the GemStone object server. Topaz is especially useful for repository administration tasks and batch mode procedures. Because it is command driven and generates ASCII output on standard output channels,

Topaz offers access to GemStone without requiring a window manager or additional language interfaces. You can use Topaz in conjunction with other GemStone development tools such as GemBuilder for C to build comprehensive applications.

UserActions (C callouts from GemStone Smalltalk)
UserActions are similar to user-defined primitives in other Smalltalks. GemBuilder for C can be used to write these user actions, and add them to and execute them from GemStone Smalltalk.

More information about the GemBuilder and Topaz products are found in their respective reference manuals. UserActions are discussed in the *GemBuilder for C* manual.

Gemstone Sessions

All of the GemStone interfaces provide access to GemStone objects and mechanisms for running GemStone methods in the server. This access is accomplished by establishing a session with the GemStone object server. The process for establishing a session is tailored to the language or user of each interface. In all cases, however, this process requires identification of the GemStone object server to be used, the user ID for the login, and other information required for authenticating the login request.

Once a session is established, all GemStone activity is carried out in the context of that session, be it low-level object access and creation, or invocation of GemStone Smalltalk methods.

Sessions allow multiple users to share objects. In fact, different sessions can access the same repository in different ways, depending on the needs of the applications or users they are supporting. For example, an employee may only be able to access employee names, telephone extensions and department names through the human resources application, while a manager may be able to access and change salary information as well.

Sessions also control transactions, which are the only way changes to the repository can be committed. However, a *passive* session can run outside a transaction for better performance and lower overhead. For example, a stock portfolio application that reports the current value of a collection of stocks may run in a session outside a transaction until notified that a price has changed in a stock object. The application would then start a transaction, commit the change, and recalculate the portfolio value. It would then return to a passive session state until the next change notification.

On both UNIX and NT platforms, a session can be integrated with the application into a single process, called a **linked** application. Each session can have only one linked application.

Alternatively, the session can run as a separate process and respond to remote procedure calls (RPCs) from the application. These sessions are called **RPC** applications. PC-based platforms (VisualAge and Visual Smalltalk Enterprise) must run in RPC mode. Sessions may have multiple RPC applications running simultaneously with each other and a linked application.

2.2 GemStone Smalltalk

All Smalltalk languages share common characteristics. GemStone Smalltalk, while providing basic Smalltalk functionality, also provides features that are unique to multi-user, server-based programming.

GemStone Smalltalk provides data definition, data manipulation, and query facilities in a single, computationally complete language. It is tailored to operate in a multi-user environment, providing a model of transactions and concurrency control, and a class library designed for multi-user access to objects. GemStone Smalltalk operates on server-class machines to take advantage of shared memory, asynchronous I/O, and disk partitions. It was built with transaction throughput and client communication as chief considerations.

At the same time, its common characteristics with other Smalltalks allow you to implement shared business objects with the same language you use to build client applications. Since the same code can execute either on the client or on the object server, you can easily move behavior from the client to the server for application partitioning.

GemStone Smalltalk extends standard Smalltalk in several ways.

Language Extensions

Constraining Variables

GemStone Smalltalk allows you to constrain instance variables to hold only certain kinds of objects. The keyword `constraints:` in a class creation message takes an argument that specifies the classes the instance variable will accept. Specifying a constraint is optional.

Constraining a variable ensures that the variable will contain either nil or instances of the specified class or that class's subclasses. When you constrain an instance

variable to be a kind of Array, you guarantee that it will always be an Array, an instance of some subclass of Array (such as InvariantArray), or nil.

Constraining Named Instance Variables

You specify constraints on a class's named instance variables when you create the class. The keyword `constraints:`, a part of the standard subclass creation message, takes an Array of constraints as its argument.

The following example creates a new subclass of Object with three instance variables constrained to be Strings and one to be an Integer.

Example 2.1

```
Object subclass: 'Employee'
  instVarNames: #( 'name' 'job' 'age' 'address')
  inDictionary: UserGlobals
  constraints: #[
            #[#name, String], #[#job, String],
            #[#age, Integer], #[#address, String] ].
```

In this example, `constraints:` takes as its argument an Array of two-element Arrays. The first element is a symbol naming one of the class's instance variables and the second element is a class to which the variable is constrained.

Array constructors (enclosed in brackets) are used here instead of literal arrays (enclosed in parentheses) to build the constraint.

The details of constraint specification differ for named and unordered instance variables. Chapter 4, "Collection and Stream Classes," explains how to constrain unordered instance variables.

Inherited Constraints

Each class inherits instance variables and any constraints on them from its superclass. You can make inherited constraints more restrictive in the subclass by naming the inherited instance variables in the argument to `constraints:` in the creation statement.

The following example creates a subclass of Employee in which the constraint on the instance variable *age* is SmallInteger instead of Integer:

Example 2.2

```
Employee subclass: 'YoungEmployee'
  instVarNames: #()
  classVars: #()
  poolDictionaries: #()
  inDictionary: UserGlobals
  constraints: #[ #[#age, SmallInteger] ]
  isInvariant: false.
```

YoungEmployee's other inherited instance variables, which are not listed in the `constraints:` argument, retain their original constraints.

You can only restrict an inherited instance variable to a subclass of the inherited constraint. So, in the previous example, you could not have constrained `age` to be of class Number or Array, since neither Array nor Number is a subclass of Integer.

Circular Constraints

A circular constraint occurs when an instance variable of a class is constrained to hold instances of its own class, or when each of two classes is constrained to hold instances of the other's class.

Query Syntax

Enterprise applications need to support efficient searching over collections to find all objects that match some specified criteria. Each collection class in GemStone Smalltalk provides methods for iterating over its contents and allowing any kind of complex operation to be performed on each element. All collection classes understand the messages `select:`, `reject:`, and `detect:`.

In GemStone Smalltalk, an index provides a way to traverse backwards along a path of instance variables for every object in the collection for which the index was created. This traversal process is usually much faster than iterating through an entire collection to find the objects that match the selection criteria.

A special query syntax lets you use GemStone Smalltalk's extended mechanism for querying collections with indexes. In addition, the special syntax for select blocks lets you specify a path of named instance variables to traverse during a query.

Auto-Growing Collections

GemStone Smalltalk allows you to create collections of variable length, allowing you to add and delete elements without manually readjusting the collection size. GemStone handles the memory management necessary for this process.

Class Library Differences

No User Interface

GemStone Smalltalk does not provide any classes for screen presentation or user interface development. These aspects of development are handled in your client Smalltalk.

Different File Access

GemStone class GsFile provides a way to create and access non-GemStone files. Many of the methods in GsFile distinguish between files stored on the client machine and files stored on the server machine. GsFile allows the use of full pathnames or environment variables to specify location. If environment variables are used, how the variable is expanded depends on whether the process is running on the client or the server.

Different C Callouts

GemStone Smalltalk uses a mechanism called *user actions* to invoke C functions from within methods. User actions must be written and installed according to special rules, which are described in the *GemBuilder for C* manual.

Class Library Extensions

You can subclass all GemStone-supplied classes, and applications will inherit all their predefined structure and behavior. This manual discusses some of these classes and methods. Your GemBuilder interface provides an excellent means for becoming familiar with the GemStone class hierarchy. A complete description of all GemStone Smalltalk classes is found in the *GemStone Kernel Reference*.

More Collection Classes

GemStone Smalltalk provides a number of specialized Collection classes, such as the KeyValueDictionary classes, that have been optimized to improve application speed and support scaling capability. A full discussion of these classes is found in the Collections chapter of this manual.

RC Classes

Reduced-conflict (RC) classes minimize spurious conflicts that can occur in a multiuser environment. RC classes are used in place of their regular counterpart classes in those applications that you determine may otherwise encounter too many of these conflicts. RC classes do not circumvent normal conflict mechanisms, but they have been specially designed to eliminate or minimize commit errors on operations that analysis has determined are not true conflicts.

User Account and Security Classes

UserProfile is used by GemStone in conjunction with information GemStone gathers during each session to provide a range of security and authorization services, including login authorization, memory and file protection, secondary storage management, location transparency, logical name translation, and coordination of resource use by concurrent users. This manual contains a discussion of how UserProfile is used by GemStone during a session. The *System Administration Guide* contains procedures for creating and maintaining UserProfiles.

Segment is used to control ownership of and access to objects. With Segment, you can abstractly group objects, specify who owns the objects, specify who can read them, and specify who can write them. Each repository is composed of segments. This manual provides a full discussion of segments in the Security chapter.

Both classes are described in detail in the *GemStone Kernel Reference*.

System Management Classes

GemStone Smalltalk provides a number of classes that offer system management functionality. Most of the actions that directly call on the data management kernel can be invoked by sending messages to System, an abstract class that has no instances. All disk space used by GemStone to store data is represented as a single instance of class Repository, and all data management functions, such as extent creation and access, backup and restoration, and garbage collection are performed against this class. The class ProfMonitor allows you to monitor and capture statistics about your application performance that can then be used to optimize and tune your Smalltalk code for maximum performance. The class ClusterBucket can be used to cluster objects across transactions, meaning their receivers will be placed, as far as possible, in contiguous locations on the same disk page or in contiguous locations on several pages.

Implementation of these classes is discussed in this manual. All of these classes are described in detail in the *GemStone Kernel Reference*.

File In and File Out

Smalltalk allows you to file out source code for classes and methods, save the resulting text file, and file it in to another repository. The GemStone class PassiveObject also allows you to file out **objects** and file them in to another repository. This functionality is similar to that provided by VisualWorks' Binary Object Streaming Service (BOSS) and Visual Smalltalk Enterprise's Object Filer. More information about the process is provided in this manual. A description of the PassiveObject class is provided in the *GemStone Kernel Reference.*

Inter-Application Communications

GemStone Smalltalk provides two ways to send information from one currently logged-in session to another:

GemStone can tell an application when an object has changed by sending the application a **notifier** at the time of commit. Notifiers eliminate the need for the application to repeatedly query the Gem for this information. Notification is optional, and can be enabled for only those objects in which you are interested.

Applications can send messages directly to one another by using Gem-to-Gem **signals**. Sending a signal requires a specific action by the receiving Gem.

2.3 Process Architecture

GemStone provides the technology to build and execute applications that are designed to be partitioned for execution over a distributed network. GemStone's architecture provides both scalability and maintainability. Sections describing the main aspects of GemStone architecture follow.

Gem Process

GemStone creates a Gem process for each session. The Gem runs GemStone Smalltalk and processes messages from the client session. It provides the user with a consistent view of the repository, and it manages the user's GemStone session, keeping track of the objects the users has accessed, paging objects in and out of memory as needed, and performing dynamic garbage collection of temporary objects. The Gem performs the bulk of commit processing. A user application is always connected to at least one Gem, and may have connections to many Gem. Gems can be distributed on multiple, heterogeneous servers, which provides distribution of processing and SMP support. The Gem also offers users the ability to link in user primitives for customization.

Stone Process

The Stone process is the resource coordinator. One Stone process manages one repository. It synchronizes activities and ensures consistency as it processes requests to commit transactions. Individual Gem processes communicate with the Stone through interprocess channels. The Stone:

- coordinates commit processing,
- coordinates lock acquisition,
- allocates object IDs,
- allocates object Pages,
- writes transaction logs.

Shared Object Cache

The shared object cache provides efficient retrieval of objects from disk, and the ability for multiple Gems to access the same object. When modified, an object is written to a new location in the cache. Memory is managed and allocated on a page basis. The cache also contains buffers for communications between Gems and the Stone. The shared cache monitor initializes the shared memory cache, manages cache allocation to the sessions, and dynamically adjusts this allocation to fit the workload. It also makes sure that frequently accessed objects remain in memory, and that large objects queries do not flush data from the cache. These controls allow complex applications to be run on the same repository by multiple users with no degradation in performance.

Scavenger Process

The scavenger process dynamically reclaims space used by unreferenced objects. This process is sometimes called dynamic garbage collection, and in GemStone, may be referred to as the GC Gem. The scavenger process also dynamically defragments the repository while maintaining requested object clustering. It has a multi-level collection architecture, consisting of:

- Dynamic cleanup of temporary objects,
- Epoch cleanup of shared objects, and
- Full sweep of the repository.

Extents and Repositories

Extents are composed of multiple disk files or raw partitions. A repository, which is the logical storage unit in which GemStone stores objects, is actually an ordered file of one or more extents. Extents can be distributed to heterogeneous servers. Objects can be clustered on an extent for efficient storage and access.

Extents can be mirrored for improved fault tolerance. By mirroring extents, you store each object in two places to reduce the chance of data loss. GemStone automatically stores each newly committed object in both locations. Any damage to one extent leaves all the objects intact in the mirrored extent, allowing GemStone to automatically switch over to the active mirrored extent on an extent fault. Using mirrored extents can also improve distributed query performance. GemStone allows the creation of one mirrored extent for each extent in the repository.

Transaction Log

GemStone's transaction log provides complete point-in-time roll-forward recovery. The tranlog contents are composed by the Gem, and the Stone writes the tranlog using asynchronous I/O. Commit performance is improved through I/O reduction, since only log records need to be written, not many object pages. In addition, the object pages stay in memory to be reused. Log files may also be mirrored for fault tolerance. GemStone supports both file based and raw device configuration of tranlogs.

NetLDI

In a distributed system, each machine that runs a Stone monitor, Gem session process, or linked application, or on which an extent resides, must have its own network server process, known as a NetLDI (Network Long Distance Information). A NetLDI reports the location of GemStone services on its machine to remote processes that must connect to those services. The NetLDI also spawns other GemStone processes on request.

Login Dynamics

When you log in to GemStone, GemStone establishes for you a logical entity called a GsSession, which is comparable to an operating system session, job, or process. GemStone creates a separate instance of GsSession each time a user logs in, and it monitors, serves, and protects each session independently.

You can log into GemStone through any of its interfaces. Whichever interface you use, GemStone requires the presentation of a user ID (a name or some other

identifying string) and a password. If the user ID and password pair match the user ID and password pair of someone authorized to use the system, GemStone permits interaction to proceed; if not, GemStone severs the logical connection.

The system administrator (or a user with equivalent privileges) assigns each GemStone user an instance of class UserProfile, which contains, among other information, the user ID and password. GemStone uses the UserProfile to establish logical names and default locations, resolve references to system objects, and perform similar tasks. The system administrator gives each new UserProfile appropriate customized rights, and stores it with a set of all other UserProfiles in the set AllUsers.

You can obtain your own UserProfile by sending a message to System. Class UserProfile defines protocol for obtaining information about default names, privileges, and so forth. More information about UserProfile is provided in this manual. Class UserProfile is described in the *GemStone Kernel Reference*, while procedures for creating and maintaining UserProfile are found in the *GemStone System Administration Guide.*

The GemStone system administrator can also configure a GemStone system to monitor failures to log in, to note repeated login attempts, and to disable a user's account after a number of failed attempts to log into the system through that account. The *GemStone System Administration Guide* describes these procedures in greater detail.