A Study on Fine-Grained Replications of Distributed Java Applications

Thepparit Banditwattanawong

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

Department of Informatics
School of Multidisciplinary Sciences

The Graduate University for Advanced Studies
Tokyo, Japan

March 2007

Acknowledgements

I wish to express my gratitude to the advisory commitee members, Professors Katsumi Maruyama, Nakajima Shin, Ichiro Satoh, Associate Professors Chiba Shigeru, Kodama Kazuya, Assistant Professors Soichiro Hidaka, and Hironori Washizaki, for constructive comments and professional suggestions that broaden my research knowledge and perspective.

Especially to my advisor, Professor Katsumi Maruyama, I appreciate his sophisticated research guidance, warm attitude, and assistance in my admission to this university.

Special thank is extended to Assistant Professors Soichiro Hidaka and Hironori Washizaki for patient and diligent comments on all of my research manuscripts and presentations and for numerous discussions throughout the regular meetings and E-mail correspondences.

Importantly, I would like to thank National Institute of Informatics (NII) and Japan Student Services Organization (JASSO) for the scholarships and Tokyo International Exchange Center for a perfect living and study environment. I am also thankful to the university's staffs for all supports of my study.

I would like to thank my colleagues, Ryota Ozaki for great assistance in several of my research experiments and Huda Md. Nurul for being a friendly consultant during my stressful times. The other friends whom I have got to know during my Ph.D. study fulfiled my maturing life.

I must also thank the pervious researchers in my field of study. The contributions of my research are derived from nothing other than standing on the shoulders of these giants.

Beyond any expression, I am deeply grateful to my beloved parents for their constant and unconditional love. Without them this work would never have come into existence.

Abstract

In distributed object systems, object-oriented (OO) applications are replicated from remote servers to client sites to improve performance, scalability, and availability. This study focuses on fine-grained replications of distributed OO applications. Unlike the traditional replication scheme by which a self-contained application is replicated entirely at once, the fine-grained replication scheme enables partial and on-demand incremental replications of self-contained applications.

Fine-grained replications can be classified into two categories based on their deployment patterns: 1) replicating running applications for local accesses and 2) downloading application programs from persistent repositories for local executions. Based on the classification, the study has proposed a pair of fine-grained replication middlewares: one aims for the fine-grained replications of remote runtime applications, and the other aims for the partial and on-demand incremental downloadings of application programs.

In addition, to exploit the fine-grained replications effectively requires a proper means to figure out application portions as the units of replication. The study has proposed object class clustering algorithms to support the use of the latter middleware, while showing that object clustering, which is used to support the former middleware, can be performed based on programmer's application knowledge.

The details of the middlewares and the class clustering algorithms are summarized individually as follows.

Fine-grained replication of runtime application: Replicating remote application objects to user locality is a common technique to reduce the effects of network problem. The traditional replication scheme is not

suitable for cooperative applications because only part of a shared application rather than a whole application should be replicated. Furthermore, the scheme is not appropriate for mobile computing devices due to their common constraints of memory spaces. Both problems can be addressed by using a fine-grained replication scheme by which the portions of a self-contained application can be replicated.

Since most object replication systems exploit the traditional replication scheme, to fulfil fine-grained replication is an unexperienced task for several application programmers. There exist few middlewares that support runtime fine-grained replications of OO applications. All of them aim for peer-to-peer applications in which objects that constitute a self-contained application are decomposed and distributed among peers. Therefore, peers that hold master copies of the application objects must always be reachable by other peers to replicate the master copies. This is not suitable for pervasive collaboration because the servant peers (e.g., mobile users) can get disconnected arbitrarily or be unreachable due to network partitioning. Instead, using dedicated servers to maintain the master copies of applications is more appropriate. Unfortunately, no fine-grained replication middleware is designed for a client-server model.

This dissertation presents SOOM, a Java-based middleware for pervasive client-server cooperative applications. SOOM provides fine-grained replication capability for clients in wide-area networks or on the Internet and allows clients in local area network to exploit a conventional remote method invocation mechanism in coordination with the fine-grained replication. SOOM also supports fine-grained concurrent access control and update synchronization. To realize the middleware, several challenge research problems have been identified and resolved.

An application for cooperative software modeling has been developed to assure the practical applicability of SOOM and demonstrate the practicality of fine-grained replication scheme, fine-grained consistency maintenance, and the coexistence of fine-grained replications and remote method invocations in client-server environment. The quantitative properties of SOOM were measured through the following empirical evaluations. First, experiments in single-user and multi-user environments based on different

consistency protocols indicated the practical throughputs of SOOM-based application. Second, throughout accessing all member objects of a benchmark cluster showed that SOOM-based replication began to outperform Java RMI when each object was accessed locally more than twice. Third, an experiment using the varied numbers of client processors assured the scalability of SOOM. Finally, an experiment on the memory space requirement showed that SOOM could reduce the significant amount of client memory space consumption as well as network bandwidth.

Fine-grained replication of application program: OO applications have been distributed more and more over the Internet. Deploying an application by retrieving the entire program from a remote repository such as HTTP server often encounters extended delay due to network congestion or large program size. Many times system resources, such as network bandwidth and client memory space, are also wasted because users do not utilize every component of the downloaded applications. Moreover, downloading a whole program at once is usually impractical for mobile computing devices due to their memory space constrains.

These problems can be addressed by decomposing a program into groups of classes and data resources to be downloaded on demand.

This dissertation presents C^2, a Java-based middleware by which a Java application can be partially and on-demand incrementally deployed via HTTP. The middleware also supports application caching and transparently automatic updating.

The launching delay of an experimented application was found to be reduced by 83% from that of the traditional whole-at-once application deployment scheme. Total program deployment and execution overhead was 22% less than that of Java Web Start.

Object class clustering approach: It is typical that only part of whole program code is necessary for successful execution. Decomposing an OO program into clusters of closely relevant classes and data resources for on-demand incremental loading optimizes the program start-up latency and system resource consumption. The lack of systematic yet simple class clustering approach prohibits this kind of optimization.

This dissertation presents a Java class clustering approach that is capable of improving both spatial locality and temporal affinity of the optimized programs. The approach provides two clustering algorithms: initial delay-centric and intermittent delay-centric ones, to achieve different requirements of optimizations.

Experimental results indicated that the algorithms were practically useful to both interactive programs and non-interactive programs. Among the tested Java programs, using the initial delay-centric algorithm and the intermittent delay-centric algorithm improved initial program loading latencies on average by 2.9 and 2.2 times respectively faster than the traditional whole-at-once program loading scheme. The intermittent delay-centric algorithm reduced the number of intermittent delays to half of the initial delay-centric algorithm. Both algorithms also led to the chances to economize on system resources, such as memory spaces and network bandwidths.

Refereed Publications

The refereed publications produced from this dissertation are listed below.

Journal

1) T. Banditwattanawong, S. Hidaka, H. Washizaki, K. Maruyama, **Cluster Replication for Distributed-Java-Object Caching**, IEICE Transactions on Information and Systems, vol.E89-D, no.11, pp.2712-2723, Nov. 2006.

2) T. Banditwattanawong, S. Hidaka, H. Washizaki, K. Maruyama, **Optimization of Program Loading by Object Class Clustering**, IEEJ Transactions on Electrical and Electronic Engineering: special issue on Electronics, Information and Systems of 21st Century, vol. 1, issue 4, pp.397-407, John Wiley & Sons, Nov. 2006.

3) T. Banditwattanawong, S. Hidaka, H. Washizaki, K. Maruyama, **SOOM: Scalable Object-Oriented Middleware for Cooperative and Pervasive Computing**, IEICE Transactions on Communications: special issue on Networks Software, vol.E90-B, no.4, Apr. 2007.

International conference

4) T. Banditwattanawong, K. Maruyama, S. Hidaka, and H. Washizaki,

Contemporaneity-conscious Clustering Algorithm for Distributed Object Caching, Proc. of the Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'05), pp. 277-283, Jun. 2005, Lasvegas, U.S.A.

5) T. Banditwattanawong, K. Maruyama, S. Hidaka, and H. Washizaki, **Proxy-and-hook: A Java-based Distributed Object Caching Framework**, Proc. of IEEE Int. Conf. on Industrial Informatics, Aug. 2005, Perth, Australia.

6) T. Banditwattanawong, S. Hidaka, H. Washizaki, and K. Maruyama, **Fine-grained Replication for Private-workspace and Memory-constrained Computings**, Proc. of IFIP Int. Conf. on Network and Parallel Computing (NPC 2006), pp. 84-92, Oct. 2006, Tokyo, Japan.

7) T. Banditwattanawong, H. Washizaki, S. Hidaka, and K. Maruyama, **Partial and On-demand Incremental Deployment of Java Application Program over the Internet**, Proc. of IEEE Int. Symposium on Communications and Information Technologies (ISCIT 2006), Oct. 2006, Bangkok, Thailand.

Contents

List of Figures

LIST OF FIGURES

List of Tables

LIST OF TABLES

Chapter 1

Introduction

1.1 Background

Distributed system provides a single coherent environment of computers and softwares to accommodate remote resource sharing. Depending on the organization of processes, distributed system can be classified into two models: client-server and peer-to-peer. In the basic client-server model, processes are divided into two groups: a server is a process implementing a specific service, and a client is a process that requests a service from a server. In contrast, processes in the peer-to-peer model can play both roles of server and client, and are hence called peers. Most distributed systems are based on the client-server model.

Object-oriented (OO) model is a widely used software abstraction methodology that encompasses the concepts of encapsulation, inheritance, and polymorphism. State (data) and behavior (methods used to perform operations on the data) are encapsulated into a single entity called an object. The key power of the OO model lies in its separation of interface from implementation.

The integration of distributed system and OO model leads to distributed object computing (DOC). Objects are distributed across network computers and typically communicate with one another by means of remote method invocation in which client objects invoking methods on and subsequently receiving responses from remote servant objects. The methods are made available to clients via a public interface. Both client's invocations and server's responses are conveyed in form of messages exchanged across the network.

1. INTRODUCTION

With the advancement of networking technologies, DOC systems have been implemented more and more on wide area networks and the Internet. However, to achieve this is not easy because network problems, such as long communication latencies and unpredictable network disconnections, are rudimentary to the large-scale DOC. They are commonly addressed by an object replication technique. Replicating objects from remote servers to client machines for local executions reduces user response times. Replication also improves system scalability when several replicas are executed on multiple client machines simultaneously. Moreover, replication enhances availability as the clients are able to request services from local valid replicas even though network connections to the servers are not available.

Most existing DOC systems exploit a traditional replication scheme, called *coarse-grained replication*, by which a self-contained OO application is replicated entirely at once (2; 12; 17; 41; 46; 51; 57). Nonetheless, this scheme is not suitable for all kinds of DOC especially those characterized by cooperative computing or pervasive computing.

- In cooperative computing such as computer supported cooperative work (CSCW), users are normally in charge of the certain portions of a single shared task rather than the entire task. The shared task is implemented as a servant application running on a central server machine to which the cooperative users or client machines connect for accessing the servant application in a relative isolation. For example, in a large-scale cooperative CAD application, a developing system model (servant application) maintained on a server is constituted from several subsystem models. Different team architects develop different certain groups of subsystem models. In other words, the architects have their own private workspaces (subsystem model) in a shared workspace (system model). This means that each cooperative client accesses only the specific portions of the servant application rather than the entire application. In this common scenario, the coarse-grained replication scheme is not appropriate since it does not support the replication of an individual application subsystem.

- Pervasive computing enables users to access the remote resources from anywhere by using small mobile computing devices, such as cellular phones, personal digital assistants. However, they usually have limitations of memory space. This often makes the utilization of the coarse-grained replication scheme infeasible:

mobile users cannot whole-at-once download the large-sized application programs for local executions. For similar reason, it is impossible for mobile clients to participate in the CSCW based on coarse-grained replications. Also, because the mobile computing devices typically have slow Internet connections, it thus takes a lengthy time to wait for a whole program to be downloaded as in the case of coarse-grained replication.

The problems pointed out in both computing paradigms can be resolved by using an intuitive solution: a remote application should be replicated in a partial fashion, and further portions of the application should be replicated later when needed. This alternative replication scheme is referred to as a *fine-grained replication*. It enables the partial and on-demand incremental replications of a self-contained application. The fine-grained replication embraces the basic advantage of the traditional scheme (i.e., the prevention of network problem effects for better performance, scalability, and availability) and new advantages, which are the abilities to realize private workspaces in client-server CSCW and reduce system resource requirements (e.g., client memory space and network bandwidth) in both cooperative and pervasive computings.

Fine-grained replications are effective for at least two application domains.

- The proliferation of mobile computing devices has been increasingly encouraging the anywhere collaboration among people. As a consequence, the client-server CSCW aforementioned has been evolving to *pervasive client-server CSCW*. In the pervasive client-server CSCW, clients in local area network access a remote servant application by using a conventional remote method invocation mechanism, while other clients in wide area networks or on the Internet replicate the servant application to their proximities to avoid network problems. The replication must achieve the notion of private workspace irrespective of whether the clients are stationary or mobile with limited memory spaces. This requirement can be met naturally by using a fine-grained replication. The pervasive client-server CSCW is a vision and one of the target application fields of this study.

- The other application field is *the deployment of easy-to-launch OO application programs over the mobile Internet* to minimize end user's effort. This deployment paradigm has been becoming more and more common. The easy-to-launch

application programs, once downloaded, can be launched directly without complete installations (just like the idea of Java applet). This deployment paradigm can be practical for mobile computing devices by using a fine-grained replication technique to hide their memory space constraints. The fine-grained replication also helps improve an initial program response time because the program can be launched as soon as its start-up program portion instead of the whole program is downloaded.

1.1.1 Types of Fine-grained Replications in DOC

As previously exemplified by the cooperative CAD system, one common usage pattern of fine-grained replication is that a running servant OO application is partially and on-demand incrementally replicated in client localities. Objects that constitute the application can be either stateful or stateless. The stateless objects do not preserve the object data across method calls unlike the stateful objects. Replicating the runtime stateful objects requires that not only their codes but also state data be replicated, whereas replicating the runtime stateless objects requires only object code to be replicated.

The other usage pattern of fine-grained replication is the partial and on-demand incremental downloadings of an OO application program (codes and data resources) from a remote persistent repository for local initialization and execution. Note that though this deployment model has no remote method invocation sent between objects located on different computers, this is regarded into the realm of DOC from the viewpoint that this is the distribution of an OO (standalone) application across the network. This way of consideration automatically excludes from the study the non-OO application or plain WWW document downloadings in the traditional distributed computing.

Based on these applicability patterns, fine-grained replications in DOC can be categorized into two types: 1) *object-cluster replication* refers to fine-grained replication of stateful objects and 2) *class-cluster replication* refers to fine-grained replication of stateless objects or partial and on-demand incremental downloadings of program codes (class files and data resource files). Consequently, these types of fine-grained replications characterize the clustering approaches, which can be categorized into two kinds: *object clustering* and object *class clustering*.

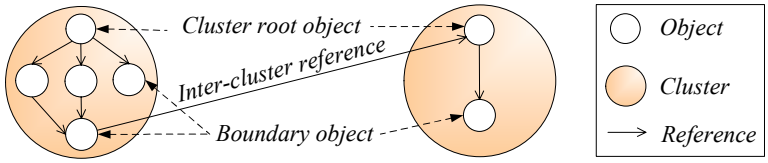


Figure 1.1: Anatomy of sample object clusters

1.1.2 Basic Concepts of Fine-grained Replication

Fine-grained replication deals primarily with the ensembles of related objects called *object clusters*. Object clusters are conceptual entities in the programming phase and actual entities in the runtime phase, i.e., the units of replication and consistency synchronization. Each cluster is constituted from an application's object *subgraph*, which consists of a single *root object*, one or more *boundary objects*, and none or several of neither root nor boundary objects (Fig. 1.1). The boundary objects delimit the group of objects in the cluster; each boundary object either holds no reference to the other objects in the same cluster or holds reference to some other cluster's object. The latter object reference is referred to as an *inter-cluster reference*. Every application object resides in some cluster. A set of dependent clusters, which belong to the same application object graph, in turn forms a *cluster graph*. The basic concepts of *class clusters* are similar to those of object clusters; the key difference is that a class cluster is constituted from a class subgraph instead of an object subgraph.

Clearly, it is essential to have clustering methods to figure out the potential groups of objects or classes to be assigned into the same object clusters or class clusters, respectively, for efficient replications.

1.2 Survey of Existing Researches

To accomplish fine-grained replication requires both a supporting platform and a clustering approach as justified earlier. The survey on previous researches related to the platform implementation is conducted in the context of Java programming language due to its several technological benefits including object orientation, platform independence, rich set of libraries, etc.

1.2.1 Fine-grained Replication Middlewares

In DOC, fine-grained replication is mostly realized in form of middlewares and rarely in form of distributed shared memory systems or distributed operating systems. *Middleware* is a software framework that provides a runtime environment beneath application and atop network operating system, realizes distribution transparency, resolves heterogeneity, and facilitates network communication and the coordination of distributed components (11; 23; 38; 56; 62). Middleware simplifies not only application deployment but also application development and maintenance.

1.2.1.1 Object-cluster Replication Middlewares

A standard middleware model, the Common Object Request Broker Architecture (CORBA), are implemented in several programming languages including Java. The body of CORBA standard offers a replication support for fault tolerance (49) by which the servant objects are replicated only in a server side without being distributed to a client side. Alhough there is the effort to provide object caching service implementations for CORBA (17; 41), the implementations are based on coarse-grained replication scheme.

Java community has defined a specification for Java object caching. Nevertheless, the notion of fine-grained replication is not present in the current specification (12). Also, its existing implementations (26; 51; 57) do not added a support for fine-grained replication. Other caching implementations of Java objects are based on the Java Remote method invocation (RMI) mechanism: (46) supports only coarse-grained replication, while (22; 45) supports partial replication but incremental replication.

As for existent Java-based middlewares supporting object-cluster replication, none of them is designed specifically for pervasive client-server CSCW. Instead, they (25; 30; 65) aim for peer-to-peer applications. This difference of application fields leads to several points of differences in the design and optimization of the middlewares. Peer-to-peer middleware platform seem to have an identical structure across all peers. It relies on push communication mechanism to achieve update propagation functionality. However, the push communication model is not practical for pervasive computing environment because user machines especially the mobile ones usually use private IP addresses and can stay disconnected anytime. Peers located in a global IP network cannot initiate the connections to other peers in private IP networks to start servicing without a special support (e.g., relay server). Push communications also necessitates

a target machine to be reachable every time a communication demand for synchronization information pushing arises.

The middleware platform for client-server model needs not have an identical structure. Instead, the structure should be optimized according to its particular role (either server or client) at runtime. As a consequence, the client-server middleware has two-tier structures: 1) a server-side middleware has one of the design goals to maintain an up-to-date central servant application and 2) a client-side middleware must be able to capture all kinds of updates and commit them to the server. To extend the client-server middleware to a pervasive client-server middleware involves one more requirement of pull communication model conformance. Specialization of the pervasive client-server middleware further towards pervasive client-server CSCW entails a new challenge of the runtime coexistence between different object accessing mechanisms, the conventional remote method invocation and the fine-grained replication. For these rationales, peer-to-peer middleware is not effective for the pervasive client-server CSCW.

1.2.1.2 Class-cluster Replication Middlewares

Since OO application programs have been growing in size, the traditional deployment method by which the programs are whole-at-once downloaded is becoming impractical for mobile users. A recent technique of compression (33) has been exploited to reduce the program's transferred size. However, the technique seems unable to cope with the problem of memory constraints in mobile computing devices. Existing variants of partial downloading technique (14; 42) can hide the memory constraints. But they do not support incremental downloading of further program portions.

A Java-based middleware for class-cluster replication has been released as an industrial product (34) recently. It supports both partial and on-demand incremental program downloading. Based on an idea of concern separation, the middleware trades off a learning curve of a specific application descriptor format for the elimination of program modification and re-compilation. This benefit would be effective only if the program was expected to be frequently modified. The middleware lacks a support for deferred downloading of actively used classes (e.g., classes appearing in the class's field declarations); they must always reside in a start-up cluster. Thus the utility of the middleware is restricted to only the programs whose most classes are non-actively

used. Furthermore, the middleware eagerly validates all the local replicas during program start-up instead of on execution demand. This leads to the wastes of system resources when some replicas, which are updated, are never executed.

The other Java-based middleware that supports class-cluster replication (63) has the same limitation of lacking the ability to defer actively used class loading.

1.2.2 Clustering Approach

In present existence, the only algorithm for class clustering (63) exists in CORBA object caching system. The algorithm requires both static information and dynamic information, thus difficult to use. Importantly, the algorithm relies on a user-defined threshold, which is used to compute the cluster boundaries. This lack of consistent clustering principle potentially leads to inefficient results of clustering: closely relevant classes may be placed in different clusters.

A research problem remaining unsolved in the field of study is an automatic approach for object clustering.

1.3 Research Objectives

Although replication has been extensively studied in the past decades, little effort has been made on the fine-grained replication of object-oriented applications in distributed environments. According to this situation, a long-term objective of this study is to promote the research and real-world deployment of fine-grained replications in DOC systems by demonstrating several benefits of the fine-grained replications in real-world practices. To contribute to this goal, this study also makes a short-term objective to devise the novel fine-grained replication supports in form of Java-based middleware platforms and clustering algorithms in such a way to address the problems identified previously.

Even though it is possible to invent a general-purpose middleware platform encompassing two features of object-cluster replication and class-cluster replication, only either of the features is utilized in a distributed object system in practice. Realizing the different types of replications into two specific-purpose platforms yields the clean designs, performance optimizations, and decrease of prerequisite system-resource requirements.

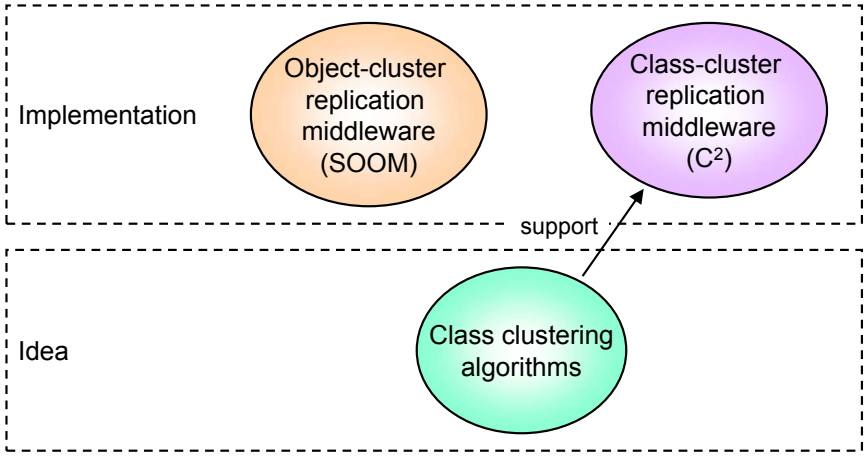


Figure 1.2: Research scope. Note that SOOM and C^2 are the given names of the middleware outcomes of this study.

Therefore, a pair of middleware platforms is devised separately. One is a middleware for Java object-cluster replication by which a runtime application is partially and incrementally replicated. The other middleware is for Java class-cluster replication by which an application program stored in persistent repository is partially and incrementally replicated. To realize these fine-grained replication middlewares must meet several requirements, such as cluster formulation, cluster realization, partial and on demand incremental executions, cluster consistency maintenance, and conceptual properties of middleware. They all have been addressed in this study.

As a counterpart of fine-grained replication middlewares, a clustering approach must be present. This is because the real effectiveness of fine-grained replications mainly depends on how good the clusters are formulated. A principle of clustering must be defined towards a distributed replication purpose. Based on the clustering principle, the two kinds of clustering schemes aforementioned in Section 1.1.1 can be invented. However, object clustering is demonstrated in this study that it can be achieved efficiently based on programmer's application knowledge. An automatic algorithm for object clustering is remained as future research to satisfy the limited time frame of the study.

Fig. 1.2 portrays an overview of research scope described above.

Chapter 2

Object Cluster Replication

2.1 Introduction

The increasing emergence of diverse pervasive computing technologies has been encouraging the collaboration in which a group of people can work in different geographical locations in relatively isolated manner by using desktop computers, laptops, personal digital assistants, and hand phones. To develop, deploy, and maintain pervasively cooperative software systems requires a considerable amount of effort because of several encountered problems related to distributed computing and mobile computing. The main problems are slow unreliable networks, platform heterogeneity, and memory space constraints in mobile computing devices. Java-based middleware seamlessly answers the heterogeneity issue. To support pervasively cooperative systems, the middleware must also address the other two fundamental issues.

Replicating objects in distributed object systems is a common technique to reduce the effects of unreliable network connections. With a traditional object replication approach, a self-contained servant application's *object graph*, which consists of a root object and typically other objects that are transitively reachable from the root object, is "entirely" replicated into client locality. This approach, however, is often not practical for memory-constrained clients. Moreover, in the context of cooperative computing, the approach does not support the notion of a private workspace in which each team participant works on a subsystem in relative isolation (i.e., only objects of certain subsystems, rather than object graph of entire system, should be replicated in clients).

Fine-grained replicating an application object graph is an intuitive solution to addressing these problems. Fine-grained replication enables partial and on-demand

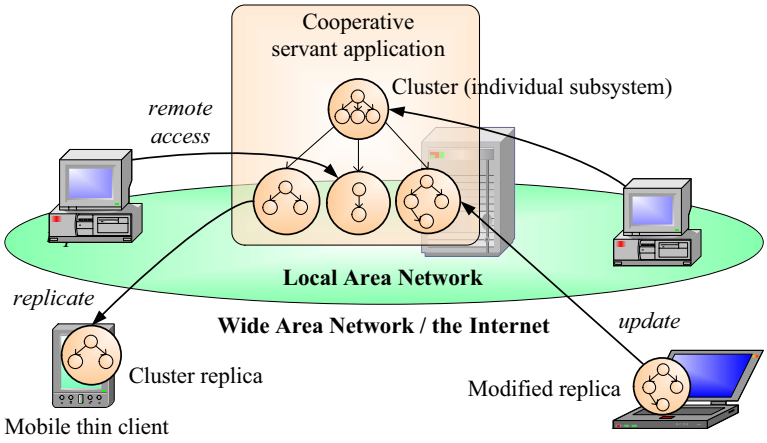


Figure 2.1: Pervasively cooperative scenario of fine-grained replication scheme and conventional remote access.

replications of a remotely shared object graph for concurrently local or disconnected processing. It economizes on client memory spaces by replicating only objects that are needed, improves an initial service response time by replicating only portion rather than entirety at first, and enables fine-grained consistency maintenance, which helps avoid the maintenance task of the unused objects.

Figure 2.1 illustrates a deployment scenario of fine-grained replications in conjunction with conventional remote access in a pervasive CSCW environment.

This chapter explains a distributed object computing platform called Scalable OO middleware (SOOM) that supports transparent fine-grained replication as a key feature. SOOM aims to liberate application programmers from system level issues, such as update synchronization, concurrency control and cooperation with a conventional remote access mechanism, by providing a comprehensive set of services in a Java context.

The intended application field of SOOM is a *pervasive client-server CSCW* as illustrated by Figure 2.1 in which participating clients in local area network perform conventional remote accesses to a shared servant application running on a central server, while stationary or mobile participating clients in wide area network or on the Internet perform fine-grained replications of the servant application. A concrete example of pervasive client-server CSCW applications is a cooperative software modeling in which team designers in different geographical locations cooperatively develop a software model maintained on a central server.

2.2 Middleware Requirement Analysis

To ensure the practicality and adequacy of SOOM, the requirements of SOOM must be captured from a realistic application. Therefore, the client-server cooperative application for OO software modeling[1] briefly mentioned above has been implemented. The application provides a shared workspace maintained on a central server for team software designers (clients) who are present in different geographical locations. The workspace is used to create a software model, a blueprint of a software system in the development and maintenance phases.

The model represents a structural abstraction of modeled software, and is constituted from a set of modeling primitives: a *diagram element*, a *package element*, and a *class element*. From a user's perspective, the diagram element represents the highest-level view of a model called a diagram. A diagram consists of packages represented by package elements. Each package may contain one or more object classes, represented by class elements, and (sub-)packages, represented by package elements (Figure 2.2 left).

Based on a Java remote method invocation (RMI) mechanism, a diagram is shared on a server by the team participants in a relatively isolated manner: each participant works on a certain remote package(s) by using GUI program. At the implementation level, the whole diagram is represented by an object graph in Figure 2.2 (right). Each object implements a certain model element. When created, objects automatically bind themselves to a Java RMI runtime. When deleted, the objects are unbound from the RMI runtime. Objects in the example object graph can be grouped into five clusters (which form a cluster graph) on a per-package basis. Each cluster has a package element as a root object and class(es) as the boundary object(s), except for a root cluster. Note that the root object `Package "consistency"` also acts as a boundary object.

At the beginning of the development process, a diagram element is created on the server. Throughout the development cycle, the participants may create new package elements (including class elements) in the diagram, remove some of them from the diagram, modifying elements' attributes, and change inter-relationships (containments) between them. These development activities, hereafter called *cluster life-cycle*

[1]The idea behind this application is identical to a commercial product that provides similar functionality for cooperative UML development (15).

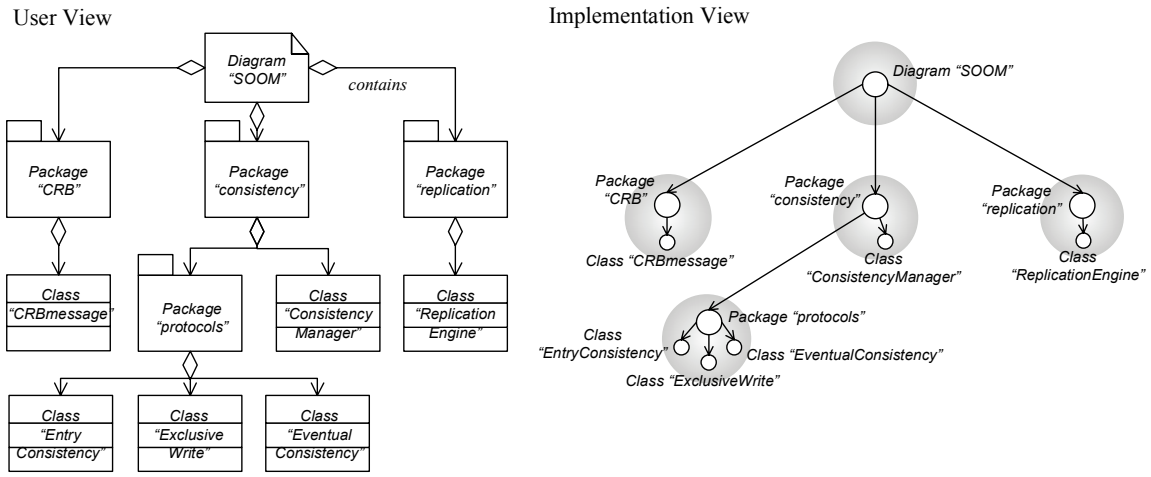


Figure 2.2: Example software model (left) and corresponding cluster graph (right).

updates, correspond to the cluster modification, creation, removal, and inter-cluster reference re-organization.

To support this operational scenario in the context of replication, SOOM must meet the following functional requirements.

1. *Support remote access-based clients*: RMI is typically used for client-server systems and especially essential for real-time cooperation. Utilizing a replication must not prohibit the use of RMI in the same system.

2. *Dynamically on-demand incremental cluster replication*: Each participant is supposed to access only the packages for which they are responsible. This means that replications should be performed at the units of clusters instead of the entire object graph. The clusters may be accessed at different times in an incremental manner, therefore, the replications need to be performed incrementally according to the runtime demands. The replication should also be transparent to the client processes to hide the intricate implementation details.

3. *Cluster life-cycle update synchronization*: Once replicated in the clients, a root cluster (representing a diagram) is used as a main entry point for the diagram to create new consecutive clusters (containing (sub)packages or classes). Some clusters may be modified, removed, or have inter-cluster references changed. These effects of these updates must be transparently reflected in an original

server-side cluster graph, which is used to serve the replication requests and RMIs.

4. *Concurrent access control*: It is imperative to take into account an update conflict among the replicas. For instance, a conflict might occur between a team member and a leader, who has authorization to edit any cluster. A consistency protocol needs to be provided to support the synchronization of concurrent accesses to replicas of the same cluster. The protocol must also be able to synchronize concurrent RMIs to a shared cluster on the server and be able to synchronize both the RMI to a server-side cluster and local access to a corresponding replica at the same time. Additionally, since the shared applications may require guarantees of different consistency semantics, fundamental consistency protocols should be provided.

2.3 Functional Design Outline

The design goal of SOOM is to meet the requirements stated above and to provide access, location, replication, and transaction transparencies (38) to application processes. Figure 2.3 illustrates the layered architecture of SOOM. SOOM relies on a client-server model in which a server-side SOOM is run on a central server machine, while a client-side SOOM can be run on several client machines concurrently to communicate with the server-side SOOM. The implementation of SOOM requires no modification in standard Java virtual machines (JVM), thus encouraging deployment ubiquity.

2.3.1 Cluster Replication

The second middleware requirement in Section 2.2 is met based on proxy and hook technique (5): *proxy* (indicated by a pentagon symbol in Figure 2.3) enables partial object graph (or cluster) replication; *hook* (indicated by a hook symbol in Figure 2.3) enables incremental replication on demand. Both operate with the assistances of a cluster replication bus, a cluster table, and a replica table. The details are as follows.

The *cluster replication bus* (CRB) is responsible for communications between the server and a client to achieve cluster replication, concurrency control, and update synchronization. The CRB uses reliable TCP/IP protocols to send or receive *CRB*

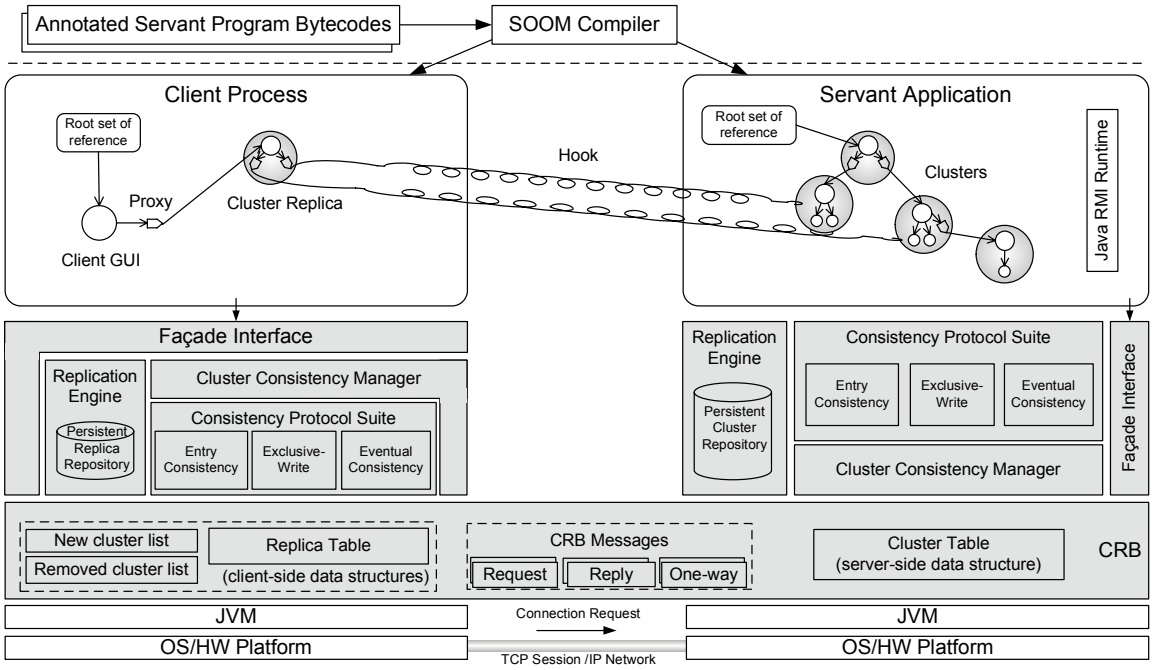


Figure 2.3: SOOM architecture.

messages, the instances of class `CRBmessage`. This class encapsulates a request, reply, or one-way data and the implementation of a `CRBmessage.execute()` method, which is used to process the data in a specific way. Example CRB messages are replication requests, replication replies, and update messages. Before dispatching, a CRB message is marshaled into a single bytecode stream. Upon receiving, the CRB message is demarshaled and its `execute()` method is invoked by a recipient CRB to process the message.

The CRB communication is conducted in a pull fashion to allow the deployment of SOOM in a pervasive computing environment. Once initialized, the server-side CRB starts listening to a supplied network socket for an incoming CRB message and spawns a new thread to fulfill each message concurrently.

When a servant application is launched, it initializes a server-side CRB and registers the application's root cluster with a *cluster table* maintained by the CRB. The table holds the pairs of CID—a unique system-wide Cluster Identification— and corresponding reference to an in-memory cluster. Once replicated in a client, a cluster replica is registered with a *replica table*—a client-side data structure, which maintains the pairs of CID and reference to an in-memory replica.

A cluster that has consecutive cluster also contains special boundary object called a *proxy*, which is responsible for holding both the reference to the consecutive cluster and its CID. Based on a Java `transient` modifier, a proxy delimits the recursion of the Java serialization process to stop at a cluster's boundary so that the cluster is a unit of a replication and update propagation. The proxy is created together with a cluster where the proxy resides and has the same program interface as the root object of a proxied consecutive cluster. However, they implement the interface differently: the proxy simply forwards the invocations to its interface to a root object of local proxied cluster for actual processing. In addition, the proxy on a client side is responsible for retaining the original program semantics, while a consecutive cluster for which the proxy is a surrogate is not yet replicated in the client locality. Without a proxy, the execution of the preceding cluster will potentially fail because it contains only an object graph fragment.

Before performing read or write accesses to a cluster (which is either local or remote), a client process must always invoke methods `CRB.beforeRead()` or `CRB.beforeWrite()`, respectively, with a reference to a local proxy of the cluster as a parameter. (Both methods are part of a facade interface to a consistency protocol API, which is described in Section 2.3.2.1.) As a result, a client-side CRB invokes the proxy's *hook()* method that utilizes a local *replication service* to replicate an up-to-date cluster from the server. The replication service creates a cluster replication request that conveys a CID supplied by the proxy and a `LastModified` timestamp of a local replica if it exists in a replica table (due to past replication) and dispatches the request message through a CRB to the server, respectively.

When a server-side CRB executes this type of replication request by invoking a `CRBmessage.execute()` method, the following sequence of actions occurs.

1. If the `LastModified` timestamp is available in the replication request, the CRB will verify the validity of the replica by comparing its timestamp with that of the requested server-side cluster.

2. If the replica is found inconsistent or there is no timestamp information available in the replication request, the CRB will consult a replication service to create the requested cluster's replica. The bytecodes of such a replica and its associated class definitions including the class definitions of the proxies representing its consecutive clusters (these class definitions are stored in a persistent cluster repository) are encapsulated in a replication reply message.

3. If the replica is found consistent, the replication reply will only contain a validity flag with a `true` value.

4. The server-side CRB serializes the reply message and delivers it to the client.

When a client-side CRB executes the replication reply message, the CRB will return the local valid replica from the replica table to the proxy if the validity flag is set true, otherwise the CRB consults a local replication service to deserialize the new replica's bytecode stream from the reply message, to store the replica into a persistent replica repository, to register the replica with a replica table, and to load the replica into the client process (the replica's reference will be held by the proxy), respectively. Now the read or write accesses (and successive ones) can be carried out locally by using the replica.

Similarly, other remote clusters that are consecutive to the local replica can be incrementally replicated on demand by invoking the `beforeRead()` or `beforeWrite()` methods on their corresponding proxies, which are available inside the local replica (Figure 2.3).

Although every inter-cluster invocation is intercepted by a proxy, this overhead is insignificant as substantiated by the experimental results; the proxy is in fact a key player in achieving cluster graph life-cycle updates (described in Section 2.3.3.1).

2.3.2 Cluster Consistency Maintenance

Maintaining cluster consistency in SOOM consists of two major tasks: concurrent access control and update synchronization (i.e., the last two requirements in Section 2.2, respectively).

2.3.2.1 Concurrent Access Control

Three consistency protocols are provided to guarantee fundamental consistency semantics at cluster granularity. The protocols are pluggable via a common interface `SOOMconsistencyProtocol`, which consists of the methods `beforeRead()`, `afterRead()`, `beforeWrite()`, and `afterWrite()` methods. These synchronization operations are client programming primitives used to control concurrent accesses and implemented based on the Java `synchronized` modifier to achieve atomicity.

As portrayed by Figure 2.4, invoking a `beforeRead()` or `beforeWrite()` may acquire read or write lock depending on the protocol and results in cluster replication (if there is no valid replica available in a replica table as described in Section 2.3.1). Invoking an `afterRead()` or `afterWrite()` may result in a read or write lock release depending on the protocol. Invoking an `afterWrite()` also sets the replica's `LastModified` timestamp and writes it back to the server before a write lock release. All critical sections are realized based on centralized locks.

1. *Entry consistency* (62): This protocol provides per-cluster critical sections to restrict concurrent write accesses to sequential ones and to prevent the sread and write accesses from occurring simultaneously (Figure 2.4 top). The protocol is suited for synchronous cooperation on the shared critical data.

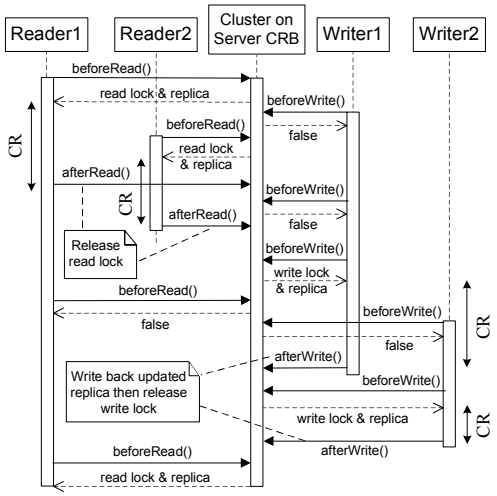
2. *Eventual consistency* (62): This protocol maximizes the degree of parallelism by allowing concurrent writes and concurrent reads at the same time (Figure 2.4 bottom). Therefore, it is suitable for an optimistic computing environments (55) where inconsistency is tolerable and conflict is occasional and easily resolvable, such as in a concurrent versions system (CVS) in which cooperation is asynchronous and seldom overlapped and personal computing environment where the only user acts as either the writer or the reader each time.

3. *Exclusive-write*: This protocol fulfils the consistency gap between Entry consistency and Eventual consistency by allowing read accesses to occur simultaneously with an exclusive write access (Figure 2.4 middle). The protocol only guarantees that concurrent writes never occur to prevent a write-write conflict, while allowing read-write conflicts (i.e., users might see some stale clusters). An example application of this protocol is a cooperative WWW-like authoring in which the team authors are controlled in a mutually-exclusive manner, and the readers prefer reading probably stale content rather than experiencing an unavailability of the content for some extended period due to authoring.

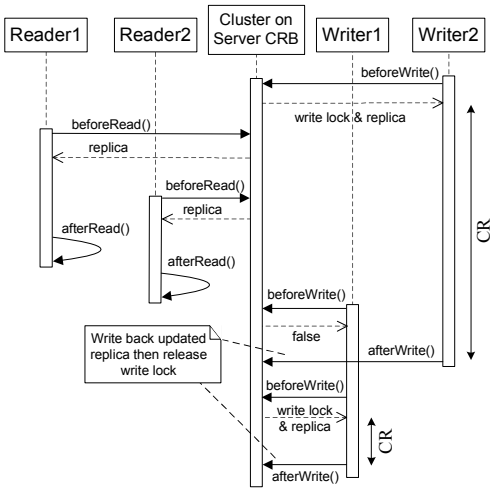
During an initialization of the server-side CRB, a servant application process plugs in a certain consistency protocol into the CRB. A client-side *cluster consistency manager* automatically plugs in the same protocol by asking a server-side cluster consistency manager.

A client who gains write permission can also perform read accesses on the replica, but not vice versa. An attempt to invoke an `afterRead()` or `afterWrite()` without

Entry Consistency



Exclusive-Write Consistency



Eventual Consistency

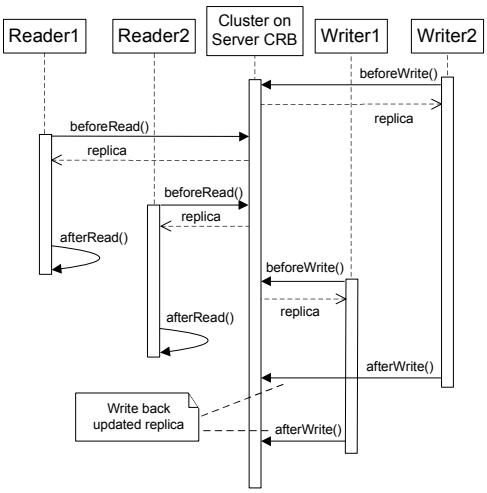


Figure 2.4: UML (50) sequence diagrams showing SOOM's consistency protocol operations. ("CR" stands for critical section.)

prior invoking a `beforeRead()` or `beforeWrite()` with respect to the same cluster results in throwing an exception `SOOMconsistencyException`. This exception is also thrown when invoking either a `beforeRead()` or `beforeWrite()` consecutively with respect to the same cluster. It is however the programmer's responsibility to ensure that a cluster does not remain locked forever. Currently, SOOM alleviates this concern by a providing `CRB.shutdown()` method to release all the acquired locks for safe client termination.

2.3.2.2 Update Synchronization

The following explains how to synchronize a server-side cluster graph with a client-side one in which an update exists.

When a client process creates a new cluster, the client-side CRB acquires a new CID for the cluster from a server-side CRB. The newly created cluster along with its CID is then registered with a replica table, and the CID is added into a *new cluster list*.

When a client process releases a cluster from a local cluster graph, the client must explicitly inform a client-side CRB by invoking a `CRB.removeCluster()` parameterized with the CID of the released cluster so that the cluster is also removed from the replica table (to allow garbage collection). The removed cluster's CID is added into a *removed cluster list* held by the client-side CRB.

Once an `afterWrite()` is invoked, the synchronization of the client-side updates proceeds by, initially, a modified replica and client-created clusters (known by using the new cluster list) as well as a removed cluster list are encapsulated into a single update message (one-way CRB message) by a cluster consistency manager. The message is then serialized and dispatched to a server-side CRB. The removed cluster list and new cluster list are emptied after committing the updates.

When the update message is executed by the server-side CRB, it first removes the clusters specified by the removed cluster list from the cluster table. The removal procedure also includes unbinding the removed clusters from an RMI runtime.

Second, the modified replica and client-created clusters are re-built and registered with the cluster table; the modified replica will automatically replace the stale one. The updated cluster (i.e., the re-built modified replica) and newly added clusters (i.e., the re-built client-created clusters) also (re)bind themselves to the RMI runtime.

In this way, the cluster life-cycle updates (i.e., the modified replica, client-created clusters, and client-removed clusters) are reflected in a server-side cluster graph.

2.3.3 Coexistence of Fine-grained Replication and RMI

The first middleware requirement in Section 2.2 implies that simultaneous operations of RMI and fine-grained replication technologies must be supported. The following issues need to be taken into account to achieve this coexistence.

2.3.3.1 Update synchronization

Since RMI is based on client-server architecture, SOOM must conform to this architecture to ensure the coexistence with RMI. The conformance leads to two design decisions as follows.

- *Centralized up-to-date cluster graph*: In distributed cooperation, rather than propagating each updated cluster replica to other associated replication-based clients (push-based update propagation) holding replicas of the same cluster, every client-side update must always be written back to a server when invoking an `afterWrite()` method. This is because a server-side cluster graph must be used to serve RMI-based clients, and therefore, it must be kept up-to-date. The replication-based clients can then fetch the up-to-date clusters from the server (pull-based update propagation) by invoking the methods `beforeWrite()` or `beforeRead()`.

- *Inter-cluster reachability maintenance*: Since the server-side cluster graph must be used to serve RMIs, which can lead to invocations between clusters, the inter-cluster reachability in the server-side cluster graph must somehow be maintained and synchronized with the client-side inter-cluster reference updates. The inter-cluster reference update can occur when a client process reorganizes the inter-cluster reference via a parameter passing resulting in a relocation of the proxy object between the relevant clusters. Examples of inter-cluster reference updates are that a newly created cluster is inserted between two clusters, an intermediate cluster is removed from a cluster graph, or two inter-cluster references are swapped.

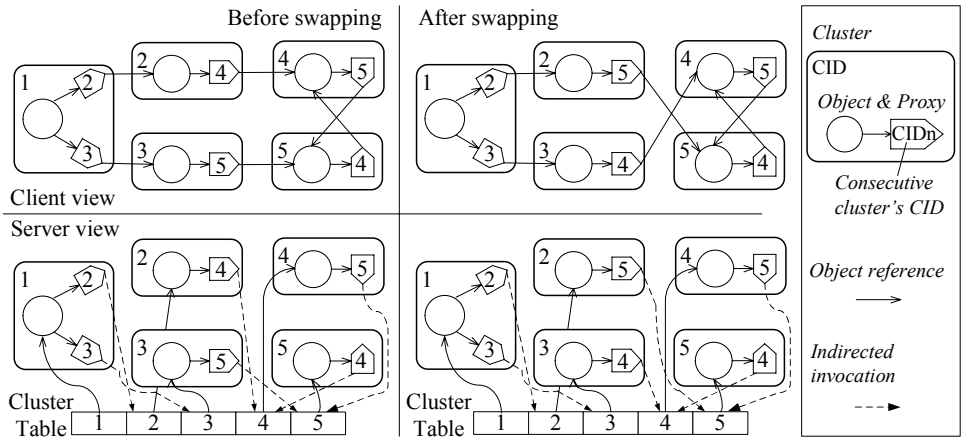


Figure 2.5: Maintaining server-side inter-cluster references.

Section 2.3.2.2 described how updates are written back to the server but did not mention about the re-building of inter-cluster references between the written back replica and other adjacent clusters in the server-side cluster graph. To maintain the centralized up-to-date inter-cluster reachability by directly updating the server-side inter-cluster references is complicated. SOOM resolves this problem by means of a proxy and cluster table-based invocation indirection.

Let us consider Figure 2.5 that exemplifies how to synchronize and realize the server-side inter-cluster reachability without manipulation of the actual inter-cluster references. Given that two inter-cluster references within a client-side cluster graph are reorganized by swapping, which results in cluster 2 pointing to cluster 5 and cluster 3 pointing to cluster 4 (client view). To reflect these updates in a server-side cluster graph, the modified clusters 2 and 3 must first written back to the server. Then, instead of manipulating the actual inter-cluster references so hardly that the new server-side clusters 2 and 3 are pointed by cluster 1 and hold the references to the clusters 5 and 4 respectively, SOOM straightforwardly replaces the references to stale clusters 2 and 5 in a cluster table with the updated cluster 2 (now containing the relocated proxy that points to cluster 5) and updated cluster 3 (now containing the relocated proxy to cluster 4) shown in Figure 2.5 the server view.

RMIs that transitively access cluster 2 followed by cluster 5 in the server view can be achieved by using a CID in the proxy pointing to cluster 5 to retrieve a

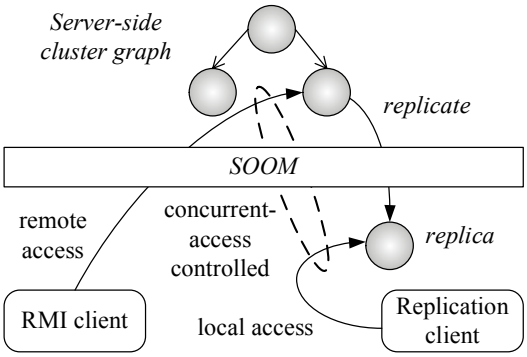


Figure 2.6: Controlling the concurrent accesses of fine-grained replication and RMI.

reference to the cluster 5 from the cluster table then the inter-cluster invocations (the RMIs) are performed through a Java reflection facility (37) based on such a reference.

With this invocation indirection strategy, the semantics of cyclic inter-cluster references (between clusters 4 and 5) shown in the figure can also be preserved.

Note that on a client side, the indirection strategy is unnecessary because actual inter-cluster references are spontaneously created through a `hook()` method during incremental replications, and directly manipulated by a client process.

2.3.3.2 Concurrent access control

When an RMI to a server-side cluster and a local access to a replica of the same server-side cluster occur simultaneously, both kinds of accesses must be controlled according to an applied consistency protocol (Figure 2.6).

Since the mutual exclusions in the Entry and Exclusive-write consistency protocols are designed based on a centralized locking mechanism (i.e., lock variables maintained on the server) as described in Section 2.3.2.1, RMI-based clients must explicitly utilize an overloading `SOOMconsistencyProtocol` API (cf. Section 2.3.2.1) so that they can issue lock management requests to the server. For example, with an Entry consistency protocol, write access to a replica is not allowed while an RMI-based client has been holding a write lock associated with the server-side cluster of the replica, and vice versa. Invoking an overloading `beforeRead()` or `beforeWrite()` will never cause replication; invoking an `afterWrite()` will never cause replica write back but will set a `LastModified` timestamp of server-side cluster.

```
public interface ClassDiagramIntf extends Remote {
  public String getName() throws RemoteException;
  public void setName(String diagramName) throws RemoteException;
  public void createPackage(String packageName) throws RemoteException;
  public void deletePackage(String packageName) throws RemoteException;
  public PackageElementIntf getPackageElement(String packageName) throws RemoteException;
  public int getCID() throws RemoteException;
}

public class ProxyClassDiagram implements ClassDiagramIntf, SOOMproxyIntf, Serializable {
  private transient ClassDiagramIntf classDiagram = null;
  private int cid;

  public void hook()  { classDiagram = (ClassDiagramIntf) CRB.hook(cid); }
  public int getCID() { return cid; }
  public String getName() throws RemoteException { return classDiagram.getName(); }
  public void setName(String diagramName) throws RemoteException {
    classDiagram.setName(diagramName);
  }
  public void createPackage(String packageName) throws RemoteException {
    classDiagram.createPackage(packageName);
  }
  public void deletePackage(String packageName) throws RemoteException {
    classDiagram.deletePackage(packageName);
  }
  public PackageElementIntf getPackageElement(String packageName) throws RemoteException {
    return classDiagram.getPackageElement(packageName);
  }
}
```

Figure 2.7: Example servant application interface, platform interface, and corresponding proxy class.

As a major benefit to not only replication-based clients but also RMI-based ones, fine-grained consistency enhances the concurrency among them: with a classical consistency control mechanism, an entire shared servant object graph is typically locked, thus blocking other RMI-based users (as well as replication-based ones).

2.4 SOOM-based Application Development

To avoid introducing a new programming model to programmers, SOOM adopts a Java RMI's programming model in which a servant application provides a remote service, and a client acquires a remote reference (proxy) to the remote service to access it. Servant applications are developed like usual RMI-based ones, and then are enabled for replication by taking some modifications in the servant program to incorporate proxies. A proxy must be generated for each cluster. The program transformation and proxy generation are facilitated by an annotation-based bytecode

```
public class FGRclient {
  public static void main(String[] args) throws Exception {
    ...
    CRB.init(serverIP, serverPort);

    //*** Accessing a root cluster "SOOM". ***//
    ClassDiagramIntf diagramSOOM = new ProxyClassDiagram();
    while(!CRB.beforeWrite((SOOMproxyIntf)diagramSOOM))
      /*wait*/;
    // Begin critical section.
    diagramSOOM.createPackage("Consistency");
    ...
    PackageElementIntf packageCRB = diagramSOOM.
                                    getPackageElement("CRB");
    // End critical section.
    CRB.afterWrite((SOOMproxyIntf)diagramSOOM);

    //*** Accessing a consecutive cluster "CRB". ***//
    while(!CRB.beforeWrite((SOOMproxyIntf)packageCRB))
      /*wait*/;
    packageCRB.createClass("CRBmessage");
    CRB.afterWrite((SOOMproxyIntf)packageCRB);
  }
}

public class RMIclient {
  public static void main(String[] args) throws Exception {
    ...
    CRB.init(serverIP, serverPort);

    //*** Accessing a root cluster "SOOM". ***//
    ClassDiagramIntf diagramSOOM = (ClassDiagramIntf)
                Naming.lookup("rmi://"+serverIP+"/SOOM");
    while(!CRB.beforeWrite(diagramSOOM.getCID()))
      /*wait*/;
    // Begin critical section.
    diagramSOOM.createPackage("Consistency");
    ...
    // End critical section.
    CRB.afterWrite(diagramSOOM.getCID());

    //*** Accessing a consecutive cluster "CRB". ***//
    PackageElementIntf packageCRB = (PackageElementIntf)
                  Naming.lookup("rmi://"+serverIP+"CRB");
    while(!CRB.beforeWrite(packageCRB.getCID()))
      /*wait*/;
    packageCRB.createClass("CRBmessage");
    CRB.afterWrite(packageCRB.getCID());
  }
}
```

Figure 2.8: Example FGR-based and RMI-based client programs.

SOOM compiler.

Figure 2.7 shows example code fragments of the application described in Section 2.2. A `ClassDiagramIntf` is a servant interface implemented by the root cluster `SOOM` depicted in Figure 2.2. A `ProxyClassDiagram` defines a proxy that is a surrogate for the root cluster. This class is pre-installed in clients or dynamically downloaded during the client program start-up; other proxies that are surrogates for non-root clusters are dynamically replicated on demand. On the server side, the root cluster is created and registered with a server-side CRB as mentioned by the following example code:

```
rootCluster = new ClassDiagram("SOOM");
CRB.init(serverPort, rootCluster, new EntryProtocol());
```

Figure 2.8 shows examples of fine-grained-replication (FGR)- and RMI-based client programs. Each of them utilizes SOOM's consistency service in order to create a pair of critical sections for accessing cluster `SOOM` and its consecutive one, namely `CRB` (cf. Figure 2.2), respectively. Note that the `getPackageElement("CRB")` method in an FGR-based client program returns a reference to a proxy representing the consecutive cluster `CRB`, which is actually replicated via a following `beforeWrite()`.

As a programming caution, applications typically must provide a feature for browsing consecutive clusters. This can lead to early replications of the clusters as they are read. For example, a user wants to browse subpackages inside a current package then operates solely on a certain subpackage. To prevent early replications, the browsed information, such as element name, in the consecutive clusters' root objects should be duplicated into the proxies surrogate for the clusters. Consequently, each proxy class should provides methods used to get the browsing information and to set it and its original one (in a consecutive cluster's root object) at the same time. This task is automated by the SOOM compiler. Programmers simply annotate the desired browsing information.

2.5 Empirical Analysis

To evaluate the benefits of a SOOM platform, several benchmarks were conducted by using heterogeneous computers as shown in table 2.1. They all had J2SE 5.0 installed

Table 2.1: Computers used in the experiments.

Computer name	Hardware specification	OS specification
Computer-1	Pentium4 3.0 GHz, 512-MB RAM	RedHat Linux(kernel version 2.4.20-8)
Computer-2	Pentium4 2.8 GHz, 512-MB RAM	RedHat Linux(kernel version 2.4.20-8)
Computer-3	Pentium-M 1.2 GHz, 512-MB RAM	RedHat Linux(kernel version 2.4.20-8)
Computer-4	Pentium Celeron 2.7 GHz, 480-MB RAM	RedHat Linux(kernel version 2.6.9-1)
Computer-5	Pentium III 600 MHz, 256-MB RAM	RedHat Linux(kernel version 2.6.9-1)

and were connected through a Fast Ethernet network. The application described in Section 2.2 was used as a realistic benchmark program to create various experimental clusters. All latency results are presented in μseconds.

2.5.1 Basic Operations

Table 2.2 presents the latencies of the client-side basic operations on a cluster, containing a single `classDiagram` object, under three consistency protocols. Computer-1 and Computer-2 were used as server and client machines, respectively, throughout all experiments described in this section. The latencies in parentheses were measured with an RMI-based client. Descriptions for these results are as follows.

- Client-side CRB initialization involved CRB's internal object instantiations and CRB message exchange to plug in a consistency protocol.
- The cost of invoking a `beforeRead()` method under Entry consistency was higher than those of Exclusive-write consistency and Eventual consistency due to a read lock acquisition (besides replicating a 11-KB remote cluster).
- Although an RMI-based `beforeRead()` with respect to Exclusive-write and Eventual consistency protocols involved neither lock acquisition nor replication, there was still in total a 25 μsec overhead of multi-layered invocations in an underlying SOOM platform.

Table 2.2: Basic SOOM performances in μsec., (): RMI case.

Operation	Entry	Exclusive-Write	Eventual
A) CRB initialization: `CRB.init()`	9000 (9000)		
B.1) `beforeRead()`	9000 (3000)	6000 (25)	
B.1.1) Read lock acquisition	3000 (3000)	- (-)	
B.1.2) Cluster replication	6000 (-)	6000 (-)	
B.2) read access: `classDiagram.getName()`	0.2 (520)		
B.3) `afterRead()`	900 (2000)	25 (25)	
C.1) `beforeWrite()`	10000 (3000)		6000 (25)
C.1.1) Write lock acquisition	4000 (3000)		- (-)
C.1.2) Cluster replication	6000 (-)		6000 (-)
C.2) write access: `classDiagram.setName("SOOM")`	0.3 (520)		
C.3) `afterWrite()`	3000 (2000)		2000 (25)
C.3.1) Write lock release	800 (2000)		- (-)
C.3.2) Update dispatch	2200 (-)		2000 (-)
D) creation: `classDiagram.createPackage("CRB")`	13000 (10000)		
E) removal: `classDiagram.deletePackage("CRB")`	35 (1400)		

- The time required by a FGR-based client to perform a read access was 1040 times faster than that of an RMI. The latency of a write access as well as that of a read access included an intermediate proxy overhead.
- In reality, the cost of a `beforeRead()` and `afterRead()` can be offset by frequent reads on a replica, while the costs of a `beforeWrite()` and `afterWrite()` can be amortized by frequent writes or reads on a replica.
- Cluster creation incurred larger latency than that of cluster removal since it needed to request a new CID from the server, and in fact, was mainly influenced by the Java object creation performance.

To demonstrate a comparative performance of FGR and RMI, a 118-KB cluster consisting of one package element object and 19 class element objects was used for completely accesses via both access schemes. For each scheme of access, the following

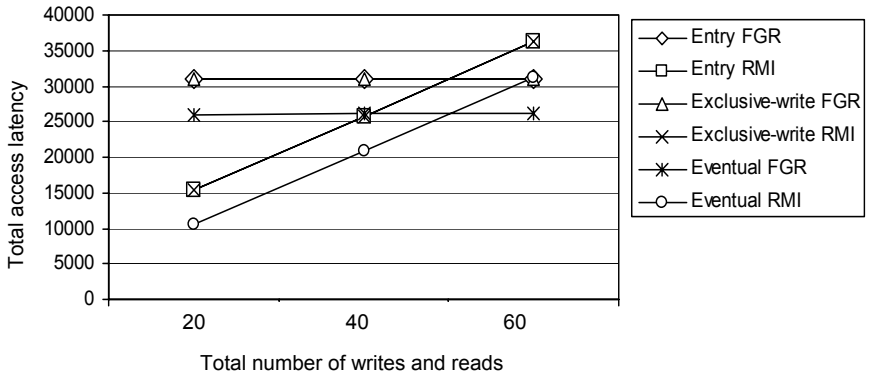


Figure 2.9: RMI and FGR cost comparison.

experimental configurations were applied: (1) each total access latency (Figure 2.9) was an elapsed time immediately before the program execution control flow entered a critical section (starting with `beforeWrite()`) until immediately exiting the critical section (ending with `afterWrite()`), and (2) the accesses were divided equally into reads and writes and distributed equally across all cluster member objects. The finding was that to outperform RMI, the cluster must be accessed at least 50 times (i.e., 3 times per cluster member object).

On the server side, the cost of an inter-cluster invocation indirection was measured by adding two timer probes into the servant code before and after a representative method `CRB.setName("communicationBus")`, which was delegated from a root cluster `SOOM` to a cluster `CRB` (Figure 2.2) by invoking a `SOOM.setPackageName("CRB", "communicationBus")` to get a `CRB` package renamed `communicationBus`. The latency of such an invocation indirection was 350 μseconds.

Figure 2.10 presents the times taken to write back newly created clusters to the server. Each of these clusters contained a single 11-KB `packageElement` object. They were created as consecutive clusters of a root cluster `classDiagram`. SOOM wrote back all of these clusters by using a single update message. Each update committing latency was an execution time of an `afterWrite()` method. The results indicated that SOOM could serve non-trivial sized private workspaces efficiently because the update committing latencies were not steeply sensitive to the numbers of committed clusters.

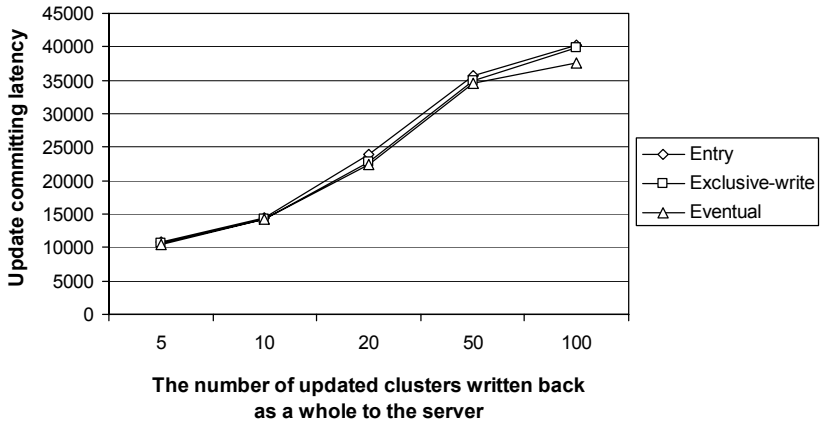


Figure 2.10: Update committing latencies of differently sized workspaces.

2.5.2 Concurrent Clients

2.5.2.1 System Throughputs

The performance of SOOM in a multi-user environment was studied in terms of overall system throughput under three consistency protocols. The experiment was set up based on a pragmatic scenario of cooperative software design described in Section 2.2: multiple software designers had simultaneous demands to access the same package (cluster) of a shared software model in server memory. A variant of Figure 2.1 in that all clients concurrently access the same cluster portrays this scenario.

Various numbers of concurrent clients were implemented as readers or writers on a per-thread basis. The clients were divided equally to run on two physical client machines (two JVMs) to generate stress workloads to the server machine (Computer-1). Since multi-user environments can range from where there are either RMI- or FGR-based clients to where RMI-based clients and FGR-based ones coexist, three experimental configurations were conducted according to the possible access means by the clients: (1) all clients were based on FGR, (2) all clients were based on RMI, and (3) half of the clients were based on FGR and the other half was based on RMI. The FGR-based clients were run on the Computer-3 client machine, and the RMI-based clients were run on Computer-2 client machine.

A new benchmark cluster that represented a moderately-sized software package (with nine member classes) was used for concurrent accessing by the above clients. The cluster consisted of one package element object and nine class element objects;

each class element had five fields and five methods. Each client acted as either a reader or a writer of the cluster. Suppose that every reader retrieved names of all cluster members for complete display on a user interface, and every writer modified every cluster member for maintenance purpose:

- Every reader invoked the same series of read operations: `beforeRead()`, `sharedPackage.getName()`, `class1.getName()` to `class9.getName()`, `class1.getFieldName(j)` to `class9.getFieldName(j)` (where field index `j` was iterated from 1 to 5 for each class element), `class1.getMethodName(k)` to `class9.getMethodName(k)` (where method index `k` was iterated from 1 to 5 for each class element), and `afterRead()`.

- Every writer invoked the same series of write operations: `beforeWrite()`, `sharedPackage.setName("foo")`, `class1.setName("bar1")` to `class9.setName("bar9")`, `class1.setFieldName(j,"foo"+j)` to `class9.setFieldName(j,"foo"+j)` (where field index `j` was iterated from 1 to 5 for each class element), `class1.setMethodName(k, "bar"+k)` to `class9.setMethodName(k,"bar"+k)` (where method index `k` was iterated from 1 to 5 for each class element), and `afterWrite()`.

Both the `beforeRead()` and `beforeWrite()` were repeatedly issued until a requested lock was granted.

Note that because the concurrent client threads share the same client-side SOOM platform at runtime, the `LastModified` timestamp comparison logic of server-side SOOM was slightly modified to ensure that the first request issued from each client thread always caused a replication of the benchmark cluster rather than reusing the benchmark replica in the replica table.

Based on the above experimental setting, Figure 2.11 presents a comparative of the overall system throughputs. Each throughput can be interpreted as the total numbers of processed read or write accesses to the server-side cluster in the case of RMI-based access and, in the case of an FGR-based access, to the client-side replicas of the servant application per second. Based on these results, the following conclusions were reached:

- For all consistency protocols, FGR could deliver higher throughputs than those of RMI in an order of magnitude because of the replication advantage.

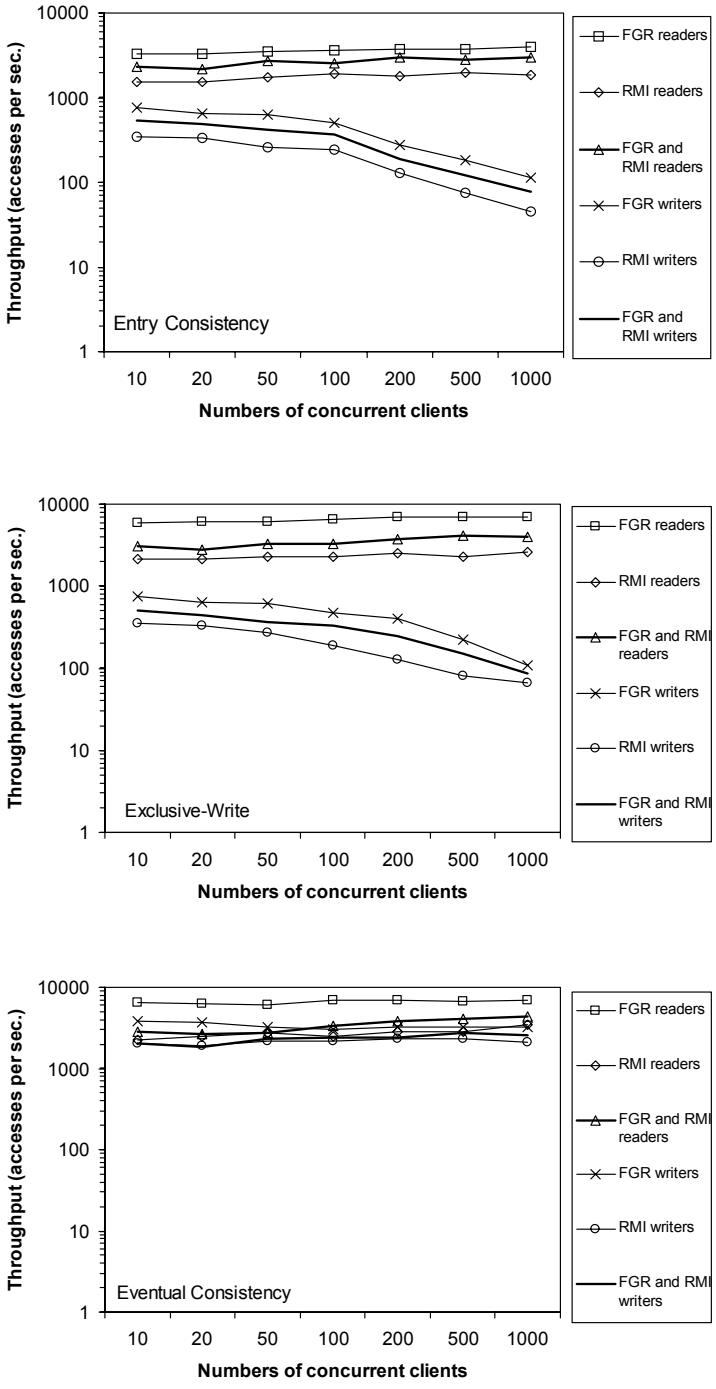


Figure 2.11: Read and write throughputs using concurrent FGR- and/or RMI-based clients under three consistency protocols: Entry consistency (top), Exclusive-write (middle), and Eventual consistency (bottom).

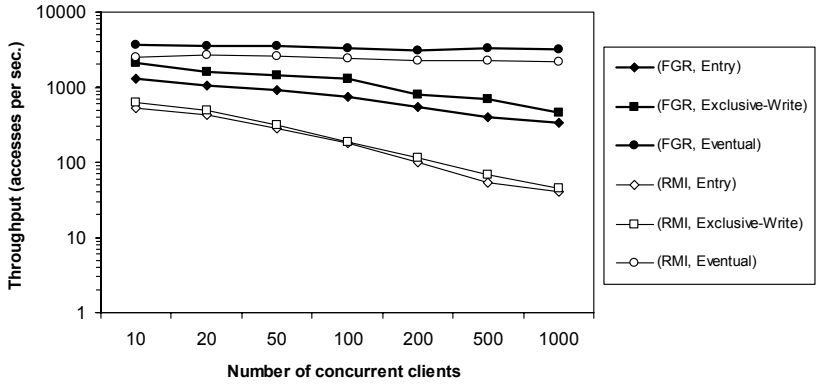


Figure 2.12: Combinative read-write throughputs of concurrent FGR- or RMI-based clients under three consistency semantics.

- Substantiated by these results, an Entry consistency traded off the slowest read and write throughputs for the strongest consistency among the other two protocols. The most efficient throughputs were gained under Eventual consistency, while an Exclusive-write consistency yielded moderate efficiency.

- FGR-based read throughputs were considerably enhanced when changing from Entry consistency to an Exclusive-write one. This implied that the overheads of a read lock acquisition and release in the Entry consistency dominated the cost of read sharing. In contrast, read lock overheads did not significantly affect the performance of RMI-based read accesses as the read throughputs only slightly improved. This implied that RMIs were expensive and dominated the cost of read sharing.

- The write throughputs based on Entry and Exclusive-write consistency protocols decreased against the increased numbers of concurrent clients. The finding from this effect was that increasing the degree of concurrency resulted in a greater overhead of the write lock management.

- The combined FGR-RMI throughputs were still more effective than those of pure RMI. These results substantiated the distinct achievement of SOOM in coordinating both FGRs and conventional remote accesses.

To study the performance of SOOM in a real-world deployment scenario, another experiment was conducted by running both readers and writers simultaneously in

equal numbers to read or write the same cluster (e.g., 100 concurrent clients consisted of 50 readers and 50 writers). Readers and writers were executed separately on Computer-3 and Computer-2 client machines, respectively. The benchmark cluster and the access numbers by the reader and writer were the same as that of Figure 2.11. The results in Figure 2.12 compared the performances of the three consistency protocols. The findings from these results were as follows.

- The combinative read-write throughputs based on Entry and Exclusive-write consistency protocols identically descended when the number of clients was enlarged. This indicated that the contribution of cluster locking overhead to the overall performances of both protocols increased by a similar rate against the increased number of concurrent clients. However, the Exclusive-write consistency protocol was more efficient due to no read lock overhead.

- FGR delivered higher throughputs than those of RMI for all tested consistency protocols.

- More general findings from these results and those presented by Figure 2.11 were that the application system's throughput was influenced by not only consistency semantics but also the read-write ratio.

The results in Figs. 2.11 and 2.12 also substantiated a possible scale of the SOOM-based application at 1000 concurrent client nodes. Precisely speaking, SOOM could accommodate the experimented servant application for up to 1000 concurrent requests without a system crash. In fact, since server-side SOOM runs on a single JVM process, the number of concurrent requests is bound to a hard-limited number of per-process file descriptors, which are used to serve simultaneous connections from clients, of an underlying operating system of a server. (Linux usually has the hard limit's default value of 1024, while allowing customization to raise this limit.) Besides the technical limitation, the practical scales of SOOM-based applications also depend on the applications' non-functional requirements. For example, the maximum number of clients can be limited by an acceptable slowest user response time because of server sharing.

2.5.2.2 System Scalaility

Theoretically, replication improves the degree of parallelism: the increased number of concurrent replication-based client processors leads to the overall system throughput improvement. To assure that SOOM is a scalable midfrastructure in practice, Computer-5 was used as a server; Computer-4, Computer-3, Computer-2, and Computer-1 were used as concurrent client machines in a respectively incremental manner (i.e., when the number of client machines was one, only Computer-4 was used; when the number of client machines was two, Computer-4 and Computer-3 were used, and so on). All clients issued the same series of accesses at an exact point of time. A benchmark cluster consisted of one package element object and 10 class element objects. Every object was read by invoking `getName()` method 10 times and written by invoking `setName()` method 10 times. The overall throughput results shown by Figure 2.13 were concluded as follows. Note that the measured elapsed time used for computing the throughputs is the time since the first client entered a critical section until the last client exited a critical section.

- For all consistency protocols, RMI-based access approach was not scalable as the system throughputs descreased against the increasing number of client processors. A rationale is that RMI-based clients must share the server computational time.

- Except for the Entry consistency protocol, FGR-based access approach was scalable as throughputs increased when increasing the number of client processors. With the Entry consistency protocol, total lock-acquisition time was proportional to the number of clients; therefore, the degree of parallelism did not improve when increasing the number of client processors. Under the Exclusive-write consistency protocol, the performance attained from concurrent read accesses outweighed system throughput drop due to sequential write accesses that resulted in the overall system scalability improvement as indicated by the graph curve's slope.

To provide precise information that helps identify SOOM-based application scales, the scalability aspect of SOOM will be elaborately studied in future work based on a scalability metric proposed in (40).

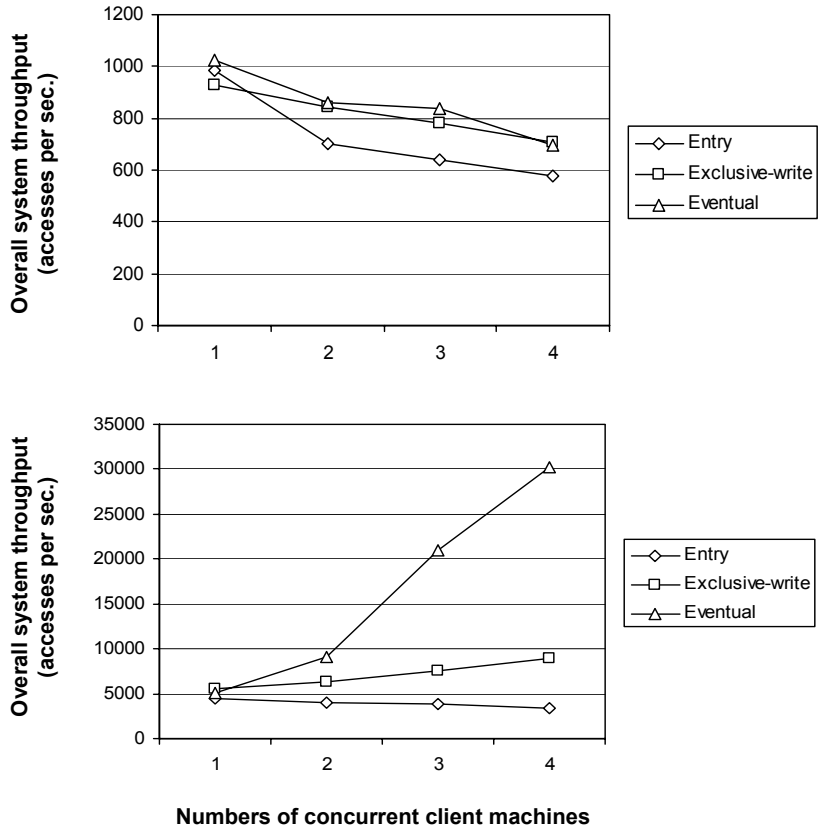


Figure 2.13: Combinative read-write throughputs using concurrent RMI-based (left) or FGR-based (right) client machines under three consistency semantics.

2.5.3 Reduced Memory Space Requirement

A new realistic object graph was constructed to represent a diagram consisting of 20 packages, each of which consisted of 10 subpackages; each subpackage also contained 20 classes (Figure 2.14). The profiled memory footprints of each kind of cluster were: 953 KB for the diagram cluster, 341 KB for the package cluster and 124 KB for the subpackage cluster. Note that the diagram cluster was largest because it contained 21 proxy objects.

Given a per-package maintenance scenario where each team participant is responsible for a single package in the above described cluster graph, this means that one diagram cluster, a single certain package cluster, and 10 subpackage clusters should be replicated in each client using FGR. Therefore, each participant requires at least 2534 KB of memory space to achieve the replication.

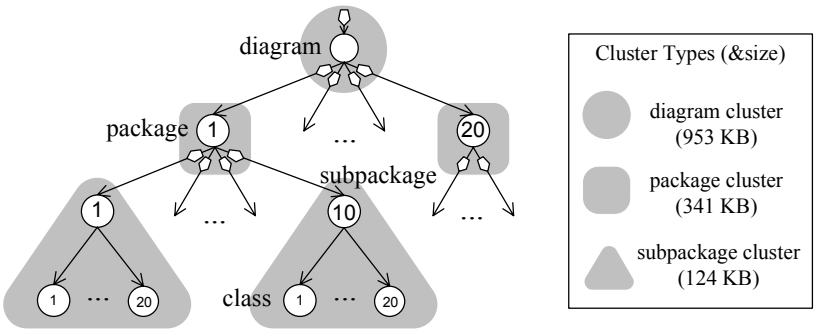


Figure 2.14: Cluster graph used to analyze clients' actual memory space requirement.

Using a coarse-grained replication approach, on the other hand, requires 25150 KB to replicate the whole shared object graph, which was not clustered. (For the non-clustered object graph, a diagram object, a (sub) package object, and a class object had the respective sizes of 129 KB, 11 KB, and 5.65 KB; no proxy objects existed in the object graph.) This memory space requirement is less practical for small computing devices with a limited memory capacity. In the per-package cooperation, network bandwidth and client memory are consumed by 19 unused packages including 190 unused subpackages and 3800 unused classes that cost 23769 KB per participant (and 475 MB for all 20 participants) in total.

In an exceptional scenario, the FGR of the whole shared object graph had 30% clustering overhead (proxy objects) in terms of size additional to 25150 KB. The coarse-grained scheme is therefore more appropriate when a whole servant application must be replicated.

With respect to SOOM-based application scales in terms of the maximum number of clusters allowed on each SOOM-based server or client, the number of clusters is not limited by SOOM's internal data structures (`java.util.Hashtable` and `java.util.ArrayList`) because of their dynamic capacities. Instead, the scales are application specific. For example, based on the measured cluster sizes shown in Figure 2.14 and a JVM's extreme heap size, the maximum number of clusters of the experimented SOOM-based servant application could be estimated: with HotSpot JVM's extreme heap size (about 3850 MB (61)), 27028 clusters (consisting of 1 diagram cluster, 2457 package clusters, and 24570 subpackage clusters) could be created.

2.6 Handling Non-replicable Objects

In reality, some objects are non-replicable because they are in any of the following conditions.

- The objects may be explicitly defined as non-replicable if they seem to consume a huge amount of system resources, contain the frequently changed content, require a central administration or maintenance, contain non-encrypted unrevealable data, or have rare access frequency (which cannot outweigh the cost of replication).
- The objects may be circumstance-dependent, such as objects containing native codes, wrapping specific device drivers, or accessing the server's file system.
- The objects may have a communication constraint: (1) They intensively interact with a non-replicable entity, such as a database front-end object that accesses a persistent data source. Replicating an object in this situation potentially increases the network traffic. (2) They are only referenced by a non-replicable object. Replicating this kind of object causes a remote communication between the non-replicable object (server side) and the replicated object (client side).
- The objects may have other technical constraints: (1) They are instantiated from a class whose some of its superclass is non-replicable. Note that we disregard the superclasses that are standard Java classes. (2) They contain threads. As our framework exploits a standard Java serialization, thread cannot be replicated. (3) They contain only static or `transient` fields. Java serialization disallows these kinds of field to be serialized.

These non-replicable objects must be anchored on the server. To enable the interactions between replicated and non-replicable objects when they are in different address spaces, an *anchored hook*, which is a special kind of hook method, can be used. The anchored hook straightforwardly exploits Java RMI to achieve the cross address-space communication.

2.7 Related Work

Most existing practices on CORBA-compliant and Java object caching (2; 12; 17; 26; 41; 46; 51; 57) do not support partial replication of the runtime object graph. The

term fine-grained replication used in (39) means per-field updating after a replica has been created by a coarse-grained replication technique. Object caching is also widely used in an OODB domain, such as Thor (44); they are typically heavyweight and out of the scope of this dissertation, which targets volatile objects rather than persistent ones.

A Java RMI-compatible implementation (22) achieves the idea of selective caching based on a *reduced object* notion. A reduced object is a version of an object where unused instance fields (as well as methods using such fields) are removed. Accesses to the instance fields or methods that are not available in the reduced object's replica are forwarded to the original object on the server. This approach does not aim for an incremental replication of an object graph.

Javanaise (13; 30) originally provides a middleware support of cluster replication for peer-to-peer cooperative applications. This means that all peers in the system should be available during the same time period to serve replication requests, thus not suitable for a pervasive computing environment. When Javanaise client instantiates a cluster by fetching its code from a coordinating server's file storage, a *proxy-out object* is created. The proxy-out object in turn creates a *proxy-in object*. When a consecutive cluster is not yet resident locally, inter-cluster invocation on the cluster will cause a cluster fault, the proxy-out object is used to locate (based on an object ID) and to fetch a proxy-in object including actual cluster. Every inter-cluster invocation is indirected via a proxy-out and proxy-in pair.

The proxy-in object is responsible for (1) fetching a consistent cluster including the proxy-out object of a consecutive cluster and (2) transparently controlling concurrent accesses to the consecutive cluster that the proxy-in object references by embedding synchronization operations in every application method exposed by the proxy-in object to block the accesses until the associated locks are granted. Therefore, every invocation on the application interface causes network communication. This practice conflicts with the idea of replication (which tries to avoid the network communications) and disallows disconnected processing. SOOM allows critical sections to be defined in client programs rather than in the proxy objects. Network communications occur only at the entrance and exit of critical section regardless of the number of invocations performed inside the critical section. Moreover, SOOM synchronization operations always return current availability status of the requested lock without any blocking to allow QoS manageability. This inside-client critical section approach

trades off the transparency of concurrent access control (or programmer comfort) for performance and QoS manageability (or end user satisfaction).

In fact, the proxy-in object is introduced in Javanaise to enable efficient and simple cluster invalidation (or updating) in peer-to-peer environment. Without proxy-in object, all proxy-out objects pointing to a to-be invalidated cluster must be identified, and invalidation messages must be sent to those proxy-out objects. It is easier to identify proxy-out object pointing to the to-be invalidated cluster, and only a single invalidation message is needed for the proxy-in object.

Because all replicas of the cluster must also be invalidated, the corresponding proxy-in objects must be available in the peers holding the replicas. However, using proxy-in objects is not suitable for pervasive client-server systems as update in a client is written back to only the server; proxy-in objects replicated in other clients are wasteful. SOOM achieves cluster validations and resolves cluster faults by applying a cluster table and a single proxy scheme (5), which is equivalent to proxy-in function but optimized for pervasive client-server systems.

When new clusters are created by methods invoked in the same critical section, they can be written back to the server by using a single update message unlike Javanaise in which a modified cluster (holding proxies to the newly created clusters) is first propagated then the requiring peers explicitly replicated the new clusters one by one using separate update messages.

Also, replacing an intermediate stale cluster with a new one in a server-side cluster graph can cause the unintentional release of valid consecutive clusters; they have to be replicated again. This problem does not appear in SOOM because references to the valid consecutive clusters are still held by the cluster table.

Javanaise does not support the coexistence between fine-grained replication and RMI. The coexistence is accomplished in SOOM by lowering and overloading the synchronization API to be utilized by both RMI- and fine-grained replication-based client programs. Moreover, Javanaise provides two consistency protocols, single-writer/single-reader and single-writer/multiple-reader, based on a broadcast communication, and uses a coordinating server to maintain global states of all replicas in the system, thus not scalable.

Manta (45) is a high-performance Java RMI implementation. It is capable of object cluster (there called *cloud*) replication but with several restrictions that limit its

applicability. Manta disallows an inter-cluster reference, thus an incremental replication is not allowed. Consistency management in Manta does not support mutual exclusion. The update propagation mechanism is based on an active replication scheme (62) whereby update operations (instead of updated objects), including parameter objects are propagated. This scheme leads to considerable computational and network overheads when clusters are frequently modified. Manta also requires a totally ordered broadcast support, thus not appropriate for a pervasive computing environment.

OBIWAN (25) is a peer-to-peer middleware that supports both incremental Java object cluster replication and RMI. It engages a dual proxy technique similar to Javanaise but differs in that the proxies are intermediate objects of two clusters in only different address spaces. A proxy-out object is used to send a cluster replication request (via Java RMI) to a corresponding proxy-in object on a remote node. Proxy-out object can also be released after a corresponding cluster is replicated. However, OBIWAN lacks a consistency protocol, which is indispensable to replication and in fact significantly influences the middleware design and implementation.

DCOBE (65) is a Java-based middleware that supports peer-to-peer cooperation based on the fine-grained replication of a *distributed composite object* (equivalent to object subgraph notion in this chapter). The composite object comprises a single *container object* (root of an object subgraph) and multiple *sub-objects* (non-root objects). The replication is achieved at the sub-object level when a read operation of the sub-object is invoked. Similar to Javanaise, the connective and control objects are inserted between the container object and each sub-object to enable sub-object replication and consistency management, respectively. DCOBE provides only an Entry consistency protocol, which can broadcast updated data or invalidation messages. Inheriting all major characteristics from Javanaise, DCOBE is not suitable for pervasive client-server CSCW.

GLOBE (62) is an object-based middleware designed for the Internet-scale distributed shared objects. Each distributed shared object, called a *local object* consists of a *semantics subobject*, which represents a real application object group, and subobjects that enable separation of concern, such as a *replication subobject*, which is responsible for replicating a semantics subobject. Although, the semantics subobject is also organized as a rooted graph (exemplified there by a group of web documents), the replication of the semantics subobject is not performed in a fine-grained manner.

The authors' previous work includes a programming framework for object-cluster replication (5) based on a single proxy scheme. It allows client-side proxy objects to be released once the remote proxied clusters are replicated. It is also able to perform on-demand incremental replication in an automatic fashion. However, the framework is partially middleware-based in terms of access transparency and replication transparency: programmers have to explicitly deal with access synchronization between RMI and fine-grained replication and the reflecting of cluster life-cycle updates in a server-side cluster graph, respectively.

Replication is also used in several software-based distributed shared memory (DSM) systems in both a page-based scheme and an object-based scheme. Recent object-based DSM systems, such as DJO (24), cJVM (1) and Hyperion (47), enables the sharing of Java objects in loosely-coupled heterogeneous systems with distribution transparency just like a middleware concept. Among these DSM systems, only cJVM supports the caching of an individual field of an object by modifying JVM. Nonetheless, the key difference of SOOM from these heterogeneous DSM systems lies in the programming model: SOOM offers a client-server model instead of a distributed shared memory one.

Besides, partial replication exists in the distributed operating system (DOS), such as E1 (54). However, DOS has a different goal from the middleware in that DOS aims for tightly-coupled homogeneous distributed systems.

Chapter 3

Class Cluster Replication

3.1 Introduction

Java is a promising technology for the Internet-based computing because of its platform independence and its rich set of libraries. Several Java applications and applets are dynamically deployed over the Internet to provide a powerful computing paradigm. However, users still encounter the same rudimentary deployment problem as in classical WWW: long program transfer delay because of large program size and network congestion. Java Archive (Jar) (33) is a recent technology that lessens this problem by compressing and packaging a program's components (class definitions and data files) into a single downloadable bundle. However, fetching a large remote program via the unpredictable QoS or mobile Internet may still result in extended user waiting time. This effect might even discourage users from using the programs that are to be dynamically deployed over the Internet. Moreover, downloading a complete program at once leads to wastes of computing resources (e.g., network bandwidth and client resources) if some program components are downloaded but do not need to be executed. This issue is particularly serious for mobile users who are charged based on transferred data amount for the network services they use.

Since not all program components are necessary for successful program execution, downloading only a program's start-up portion at first and then downloading further portions incrementally on demand is an intuitive solution to the previously stated problems. As a concrete example, a program may be decomposed into first, a start-up portion consisting of GUI class definitions and relevant program resources (e.g., configuration files) that are necessary for program launching to enter a main event loop and second, other deferrable program portions, each of which implements a specific

program function (e.g., help function). A *partial and on-demand incremental deployment* strategy improves not only initial program response time but also economizes on system resources because only the needed program portions are loaded. Combined with Jar technology, this deployment strategy becomes more effective. Consequently, the idea of *on-demand incremental program update* can be realized: in this process, each individual program portion is a unit of upgrading so that overheads in upgrading the unused program portions can be avoided.

This chapter explains a middleware called Cluster Caching (C^2) by which Java application programs can be partially and on-demand incrementally downloaded and cached at granularity of program component *cluster* (the group of relevant class implementations and data resources of a program).

3.2 Related Work

A well known implementation of Java Network Launching Protocol (JNLP) (59) called Java Web Start (JWS) (34) supports dynamic deployment of Java applications in Jar format over the Internet just like Java applets. JWS allows Java applications to be executed offline (from a desktop shortcut or cache viewer); this is unlike applet, which has to be launched from a Web page and thus cannot be used if the web server hosting the applet is down. JWS also supports partial and on-demand incremental downloading of Java applications (and applets[1]) through a lazy downloading feature, by which the deferrable application Jars are marked as `lazy` in the application descriptor. JWS-based programmers have to learn and maintain throughout application development and maintenance cycles the complicated application descriptors (JNLP files) in parallel with program source codes. Because JWS has a restriction that every application has to be launched in Jar file format, access to the application's data (non-code) resources can no longer be performed in simple ordinary style; API such as `java.lang.ClassLoader.getResource()` (60) is needed. Although JWS is convenient from the viewpoint of concern separation as using the application descriptor needs no program modification and re-compilation, this benefit might not outweigh the disadvantages of using application descriptor mentioned above if the

[1]Since an applet is typically a small-sized program, it does not gain much performance improvement from the partial downloading technique.

need for program modifications (which will occur in application descriptors) is not often.

With JWS's lazy downloading feature, *actively used classes*—ones that are loaded early by the linking process of JVM (43), such as classes used in other classes' fields, constructors and static initialization blocks—cannot be lazily downloaded and must always be included in the program's start-up portion even though they might never be executed. Lazily downloading (and caching) of data resources has to be explicitly controlled via complicated `javax.jnlp.DownloadService` APIs. JWS also provides an incremental update feature, which requires `JarDiff` and special servlet supports. With this feature, all locally cached Jar files of an application are eagerly validated during the start-up phase (rather than validated on demand) even though they might not be entirely utilized during execution; the application's JNLP file itself also has to be maintained for update.

CASCADE (63), a CORBA-compliant Java application caching system, employs multiple user-defined class loaders to enable incremental downloading of program class bundles. The class loaders do not participate in JVM's resolution process. They are instead responsible for supplying a class bytecode when a JVM encounters a reference to a class it cannot find. In other words, CASCADE cannot defer the downloadings of actively used classes.

The class splitting technique (14) partitions each Java class file into hot (frequently used) and cold (seldom used) portions based on the profiling information to lesson transfer delay. The cold portions may include actively used classes that are rarely or never executed. Transferring the hot classes can be overlapped with program execution via a pre-fetching capability. This work, however, does not support incrementally downloading a program's cold portions.

Java dynamic class loading (42) enables programmers to define their own network class loaders from scratch by subclassing `java.lang.ClassLoader`.

The authors' previous work on class cluster replication (5) provides a programming framework for partial and on-demand incremental code replication of Java program based on lazy object creation and hook techniques. That framework is extended into a middleware with cache coherence detection and enforcement capabilities as described in this chapter.

In the context of the Internet-based deployment of Java applications, this chapter presents the following contributions:

- The design and implementation of a novel *mobile object code computing middleware*[1], which has the following unique features:
  - Support for partial and on-demand incremental downloading of not only a program's data resources and inactively used classes but also its actively used classes. This capability is achieved based on a class-cluster replication framework (5).
  - On-demand incremental updating of program portions. The local copies of program fragments are neither eagerly nor entirely updated during program launching. Instead, they are incrementally updated on demand throughout program execution in response to their first uses. This feature operates automatically and transparently to programmers.
  - Ease of development. Programmers need not learn and maintain application descriptors, while data resources can be accessed via usual `java.io` APIs. Both the downloading and updating features mentioned above are achieved via a single point of service called `hook()` method.
- The comparative performance analysis of JWS, a traditional whole-at-once program downloading scheme and the proposed middleware.

3.3 Requirement Analysis

To address the problems identified above and to ensure the comprehensiveness of C^2 in supporting Internet-based application development and deployment, the following requirements must be met.

- **R1:** To alleviate the effects of large program size and the Internet's unpredictable QoS, partial and on-demand incremental program downloading must be supported. It should also be performed in a way that is transparent to end users.

[1] Although, the middleware runs on the client side only, the middleware is considered as middleware (cf. Section 1.2.1 for the definition of middleware) because it realizes replication transparency and coordinates distributed components (OO applications running on client machines) and the code stores (HTTP servers).

- **R2:** To minimize network overhead and bandwidth requirements, programs should not be downloaded at the smallest granularity of individual components one by one. Instead, the program's relevant classes and data resources should be packed together and transferred in the form of compressed Jars.

- **R3:** To speed up program launching time, the program updatings should occur incrementally based on real usage demands, and should be performed automatically and transparently.

- **R4:** To enable the program to be launched anywhere and at anytime, it must allow offline execution of local copies of program portions so that .

- **R5:** It should allow a simple application development cycle. Programmers should not need to learn or maintain any application descriptor other than their program source codes and data resources. Programmers should also be able to access the data resources with ordinary API.

- **R6:** Lastly, to allow ubiquitous deployment, it must not require any modification in a standard JVM. Furthermore, since deploying programs on the Internet is typically hindered by proxies or firewalls, all communications should rely on a ubiquitous protocol, i.e. HTTP. As a consequence, standard Web servers can be used to host C^2-based application programs.

3.4 Functional Outline

C^2 is designed to meet the above functional requirements as described below.

3.4.1 Partial and On-Demand Incremental Downloading

This section describes how to meet requirements R1, R2 & R6. Because this feature prescribes that all class definitions (and data resources) of a program need not be available together in the same address space at the same time, the program may require modification (depending on whether or not the program's classes actively use other deferred classes) so that the execution of each Jar, which contains a fragment of the program, can succeed.

```
public class TheSky implements ActionListener {
  private SolarObject sun=new SolarObject(1300,150000);
  private SolarObject moon=new SolarObject(3.5,384);
  ...
  public void actionPerformed(ActionEvent e) {
    String source = e.getActionCommand();
    ...
    if(source.equals("Sun")) {
      distance = sun.currentDistance();
      size = sun.apparentSize();
    } else if(source.equals("Moon")) {
      distance = moon.currentDistance();
      size = moon.apparentSize();
    } else if(source.equals("Usage")) {
      Help.showUsage("Usage.doc");
    } else if(source.equals("Home location")) {
      Help.showWorldMap("WorldMap.jpg");
    } else if(source.equals("Exit")) {
      System.exit(0);
    }
  }
}
```

Figure 3.1: Example interactive program

Let us consider a moderate-sized scientific program in Figure 3.1. A main class `TheSky` actively uses (instantiates through class fields) class `SolarObject` and inactively uses class `Help`. The program's deployment can be optimized by assigning the main class to a start-up Jar and other deferrable classes to separate Jars (Figure 3.2). Each Jar is titled with a uniquely arbitrary name (here, a contained root class's name).

To achieve this optimization, the program is modified (on the lines with asterisks) as in Figure 3.3.

Codes on lines 2, 9-11 and 3, 16-18 defer the instantiations of actively used class `SolarObject` by using a lazy object creation design pattern. Lines 10 and 17 utilize C^2's API namely `CRB.hook()` to download Jars (whose names are specified by `String`

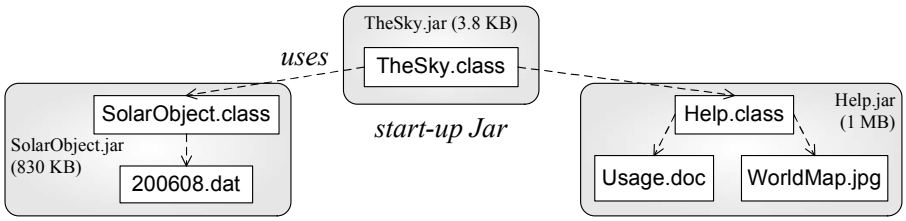


Figure 3.2: The Example Program's classes and data resources grouped into Jars

parameters as shown in Figure 3.2) containing the required class definitions from the HTTP server.

In the case of inactively used classes, the JVM do not eagerly load these classes during instantiation of `TheSky` class. However, the inactively used classes must be dynamically downloaded whenever demanded by invoking `CRB.hook()` on lines 23 and 26 immediately before their first uses. Once, `Help.jar` is cached, C^2 manages these two program statements in such a way that invoking them causes no underlying operation in C^2.

In this way, the program can be safely partitioned into different Jar files, which will be downloaded incrementally on demand. Users launch the program by explicitly downloading only the program's start-up Jar (`TheSky.jar`); other individual Jars are downloaded on demand. The downloaded Jars are stored in a certain directory, which is used as C^2's per-application cache. The Jar sizes shown in Figure 3.2 are those after modifications.

Note that to utilize `CRB.hook()`, Java event-driven programs can be written based on not only a call-back technique using `this` object reference (which is omitted in Figure 3.1) but also an anonymous inner class (which is used to create a listener class) e.g. `menuItem.addActionListener(new java.awt.event.ActionListener() {...});`. However, using the call-back technique, `CRB.hook()` might have to be invoked in many sites in a program with respect to caching the same class.

3.4.2 Transparent On-Demand Incremental Updating

This section describes how to meet requirements R3 & R6. C^2 achieves on-demand incremental updating in two phases: during program launching and during incremental downloading. First, when a program's local start-up Jar is executed, C^2 automatically validates the start-up Jar by checking with the HTTP server to determine whether

```
1   public class TheSky implements ActionListener {
2*    private SolarObject sun = null;
3*    private SolarObject moon = null;
4     ...
5     public void actionPerformed(ActionEvent e) {
6       String source = e.getActionCommand();
7       ...
8       if(source.equals("Sun")) {
9*        if(sun == null) {
10*         CRB.hook("SolarObject");
11*         sun = new SolarObject(1300,150000);
12        }
13        distance = sun.currentDistance();
14        size = sun.apparentSize();
15      } else if(source.equals("Moon")) {
16*       if(moon == null) {
17*         CRB.hook("SolarObject");
18*         moon = new SolarObject(3.5,384);
19        }
20        distance = moon.currentDistance();
21        size = moon.apparentSize();
22      } else if(source.equals("Usage")) {
23*       CRB.hook("Help");
24        Help.showUsage("Usage.doc");
25      } else if(source.equals("Home location")) {
26*       CRB.hook("Help");
27        Help.showWorldMap("WorldMap.jpg");
28      } else if(source.equals("Exit")) {
29        System.exit(0);
30      }
31    }
32 }
```

Figure 3.3: Example program after modification

or not the local copy of the start-up Jar is up-to-date or not. If not, a valid copy of the start-up Jar will be downloaded and will replace that stale copy in a cache. In this case, the program must be re-launched by using the newly cached start-up Jar.

Second, before downloading further Jars (requested via `CRB.hook()`) C^2 automatically checks in a local cache to determine whether or not a valid local copy of the requested Jar is available. If not, C^2 will transparently download and cache a valid copy from the HTTP server.

A cache coherence detection procedure is used to verify whether any update is present on the server, this procedure relies on a pair of timestamps. First, a server-side Jar's timestamp is read from a `Last-Modified` header field returned by a HTTP `GET` request from the server. Second, a timestamp of locally cached Jar (if any) is a file-system last modified attribute of a C^2-generated cache meta file associated with the cached Jar.

Every used class definition or data resource is validated only once throughout program execution even though `CRB.hook()` (Figure 3.3, lines 23, 26) is invoked every time when an inactively used class is accessed.

3.4.3 Offline Execution

This section describes how to meet requirement R4. When the application in a cache is launched without connectivity to a remote server, the local cached Jars of the application will be automatically utilized by a cluster replication broker (CRB). Network connection timeout exceptions occurring during cache coherence detection are caught and ignored to allow disconnected execution. Of course, to execute an uncached Jar requires a network connection to the server.

3.5 Non-functional Outline

3.5.1 Ease of Development and Deployment

This section describes how to meet requirement R5. All of the above functions can be readily realized through a minimal set of C^2's APIs (`CRB.init()` and `CRB.hook()`).

Once downloaded, every Jar (except the start-up Jar) is unpacked into a local cache and an associated *cache meta file*—a null file with extension `rep` automatically created for each Jar when it is cached in a local cache store; a cache meta file is

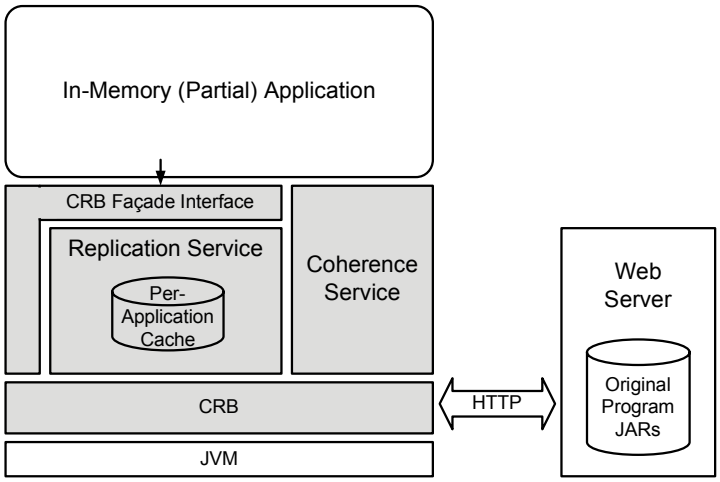


Figure 3.4: C^2's architecture

created with the same name as the cached Jar so that it can be readily identified by a coherence service. This design decision allows applications to access their local data resources in usual fashion without the inconvenient API necessitated by Jar technology. Based on the cache meta file, the cache coherence can virtually be maintained at the unit of Jar rather than by individual class definition or data resources. Each Jar should contain only relevant classes or data resources for effective cache coherence maintenance.

C^2 itself is also a lightweight middleware (size in Jar format is only 3 KB). It can be shrink-wrapped in application start-up Jars for effortless deployment by simply clicking the downloaded start-up Jars to launch the applications; of course, this requires a Jar manifest.

3.6 Architecture and Operation

Figure 3.4 illustrates C^2's architecture. Its operational steps are next described in details. They are performed transparently to application processes. Figure 3.5 summarizes these operation steps.

1. **C^2's initialization:** When a start-up Jar, which is downloaded manually by a user, is executed, the application's `main()` method invokes method `CRB.init()`

of a CRB with a supplied URL of an HTTP server and the start-up Jar's file name as the parameters. For example:

```
CRB.init(serverURL, "TheSky");
```

2. **Cache coherence detection:** CRB then verifies the validity of the start-up Jar by consulting a cache coherence service. If there is no cache meta file of the start-up Jar available in the local cache (the file directory where the start-up Jar resides) because the client is launching the application for the first time, the coherence service creates a cache meta file for the start-up Jar (e.g., `TheSky.rep`). The cache meta file's last-modified timestamp is also synchronized to be the same as that of the start-up Jar.

   Then the coherence service retrieves the last-modified information of the start-up Jar's local copy by invoking method `java.io.File.lastModified()` on the start-up Jar's cache meta file. The coherence service also retrieves the last-modified information of the start-up Jar's server-side copy via method `java.net.HttpURLConnection.getLastModified()` which is parameterized with a fully specified URL to the server-side copy.

3. **Cache coherence enforcement:** If the coherence service finds that the server-side copy is newer than the local one, it will consult a replication service to fetch a valid copy to replace the stale copy in the cache. Once cached, a new cache meta file of the start-up Jar (e.g., `TheSky.rep`) is created and the application is forced to terminate so that the validated start-up Jar can be used in a new execution.

   If no update is available on the server, `CRB.init()` returns and the application continues execution.

4. **On-demand incremental caching:** During the application's execution, if some further Jar is demanded (i.e., `CRB` receives a message, for example, `CRB.hook( "SolarObject")`) the coherence service will search in a cache for a valid copy of the requested Jar (`SolarObject.jar`). If the requested Jar's cache meta file (`SolarObject.rep`) is not present in the cache (i.e., `SolarObject.jar` is not yet cached), a server-side copy of the Jar will be downloaded. Otherwise, if the cached copy has not been validated since program launch, a valid Jar copy (verified based on timestamps as previously described) will be brought from the

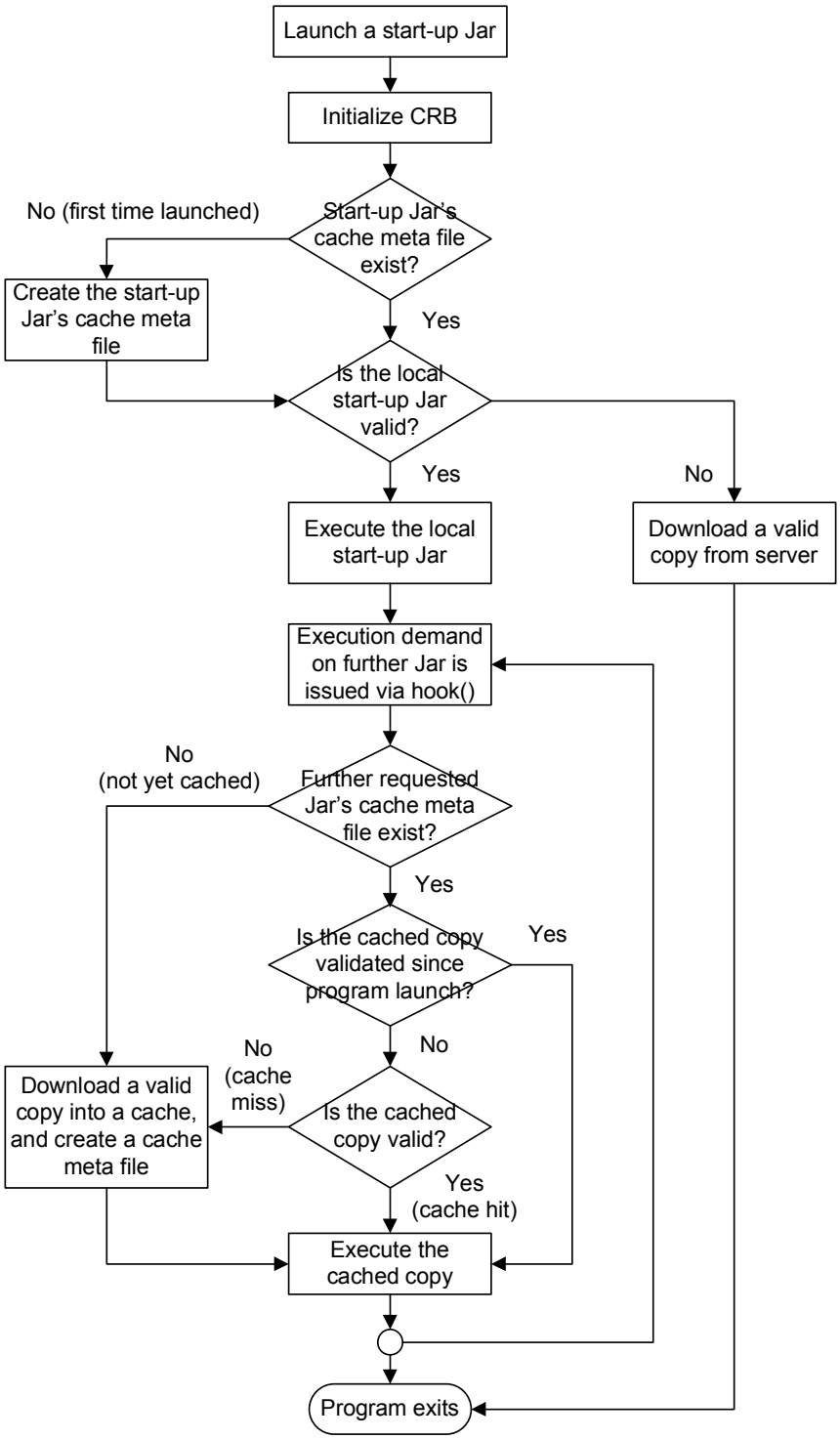


Figure 3.5: C^2's operational steps

server. The newly downloaded Jars in both cases are unpacked into the cache, and a corresponding cache meta file is created in the cache. The last-modified timestamp of the cache meta file is also synchronized to be the same as that of the downloaded Jar. The application's execution then proceeds based on the newly cached Jar.

There is a difference between caching a start-up Jar and a non-start-up Jar: the latter is fetched by invoking `java.net.JarURLConnection.getJarFile()` method and is unpacked before being saved in the cache. Because the start-up Jar is cached in the form of Jar, it should not contain any data resource (to avoid using `java.lang.ClassLoader.getResource()`) which can be instead placed in a separate Jar. This design decision enables applications to be both conveniently launched in non-command line mode and developed by using usual resource access mechanisms.

3.7 Performance Experience

To evaluate the performance of C^2, experiments were conducted by using two machines connected via an NIST Net (48)-emulated 56 Kbps network. (The 56Kbps speed is the representative data transfer rate of modems or 2.5G mobile phones, the target domains of C^2.) The HTTP server was Pentium4 3.0 GHz and the client machine was Pentium-M 1.2 GHz, both with equal 512-MB memories. Apache version 2.0.40-21 was run on top of Linux RedHat 9 (kernel version 2.4.20-8) on the server. The client used MS Windows XP Professional (version 2002 with service pack 2 installed), J2SE 5.0 (including built-in JWS), and MS Internet Explorer version 6.

The interactive program `TheSky` described previously was used as a realistic benchmark program. The performances of three deployment approaches were measured:

- The first approach used the non-modified version in Figure 3.1 together with JWS. The start-up Jar (831 KB) consisted of two classes, `TheSky` and `SolarObject` (as the latter could not be deferred for downloading in JWS), and data resource `200608.dat`. The other Jar was `Help.jar` consisting of `Help.class`, `Usage.doc`, and `WorldMap.jpg`.

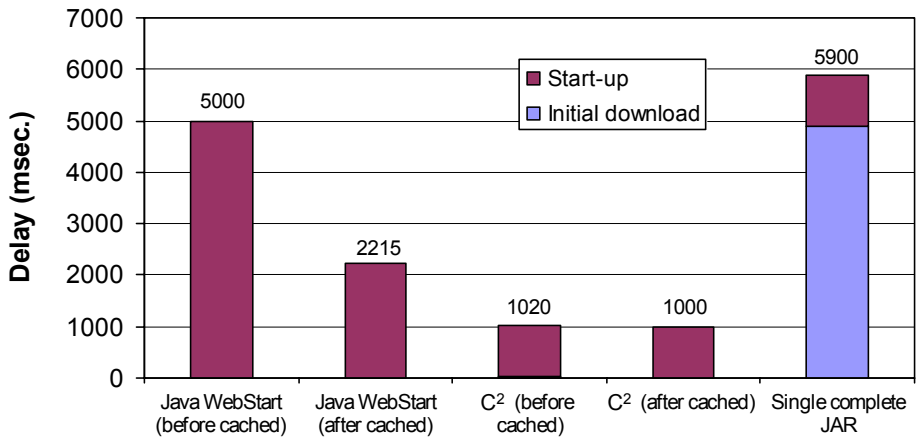


Figure 3.6: Program launching delays

- The second approach was based on C^2 and used the code in Figure 3.3. All Jars were manipulated as shown in Figure 3.2. In addition, C^2 was also packed into `TheSky.jar` (whose new size was 6.5 KB).

- The last approach, called *whole-at-once program deployment*, used the non-modified version; all class definitions and data resources were packed into a single Jar (1.84 MB) to be wholly downloaded at once with the Internet Explorer.

3.7.1 Program Launching

The earliest performance aspect that users can recognize is program launching delay. It comprises an initial downloading delay of the program start-up portion and a start-up delay in executing the start-up portion until the main loop of the event is entered. In the case of whole-at-once program deployment, program launching delay is the total time taken for downloading a whole program (i.e., a single complete Jar) and starting it up. Program launching delay seems to be a key factor to the application deployment success since users often give up downloading programs that take a long time.

Figure 3.6 compares the program launching delays of all of the deployment approaches tested. Each approach was tested in two configurations: before and after cached. In the case of "after cached" delays, the program was exited and launched

again. The delays did not involve another downloading delay because the server-side Jars were not updated. All results were measured while online. However, cache coherence detection did not operate in the case of JWS but did in the case of C^2 (because the start-up Jar must be downloaded before executed). The ratio between initial downloading and start-up delays in the case of JWS's "before cached" was not determined.

- The results demonstrated that C^2-based launching before caching was 3980 milliseconds faster than that of JWS mainly because of caching overhead (JWS-based start-up Jar, which was 827.2 KB bigger than that of C^2, took 2200 milliseconds to download) and the loading latency of JWS itself (taking approximately 1545 milliseconds[1]). As a remark, the initial download latency in the case of C^2 before cached was 20 milliseconds. C^2-based launching delay before caching was 17% of the total time taken to download and start up a single complete Jar.

- Regarding after-cached performance, C^2 still performed 1215 milliseconds faster than JWS due to the JWS's self-loading latency, the latency to early validate the start-up Jar, and the latency of dynamic start-up Jar unpacking.

3.7.2 Incremental Caching

When an interactive application is partially and incrementally deployed, the execution time of each function involves not only the function's processing time but also caching latencies (cache coherence detection and either corresponding enforcement or initial replication) of the program portion by which the function is implemented.

`TheSky` benchmark has four major functions (excluding `Exit`) as shown by the action events listened in Figure 3.1. In the JWS-based deployment approach, since the implementations of functions `Sun` and `Moon` were packed in the same start-up Jar, executing these two functions did not involve caching delay. In C^2-based deployment, executing each function incurred caching delay (for cache coherence detection and corresponding coherence enforcement). With a whole-at-once deployment approach, each execution time solely involved function processing time (and also Jar unpacking delay). Figure 3.7 presents the elapsed times in executing each function in a top-down order as shown by legend in the figure. The "after cached" execution times

[1]The latency potentially included XML parser loading latency.

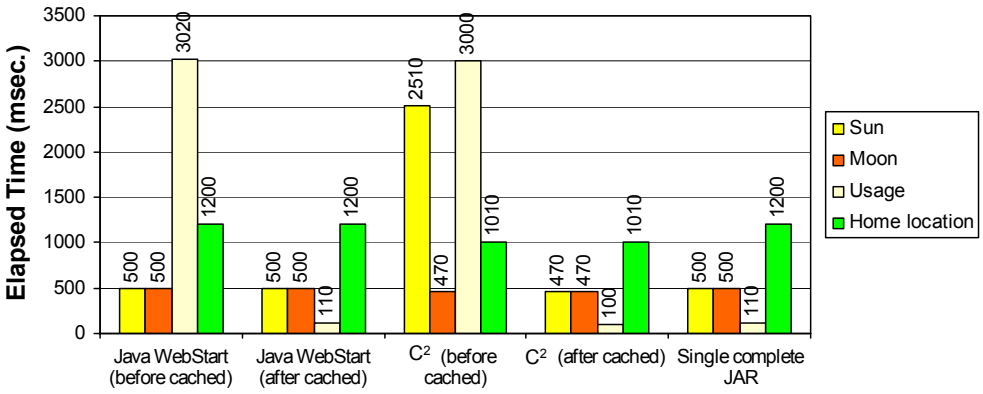


Figure 3.7: Execution times of four program functions

were measured once the program was restarted (after measuring the "before cached" execution times). From these results, the following conclusion can be reached:

- In the case of C^2 before caching, executing a function `Moon` was faster than executing a function `Sun` because a `SolarObject.jar` was already cached when executing the function `Sun`. Similarly, executing a function `Home location` was faster than that of a function `Usage` because a `Help.jar` was already cached when the function `Usage` was executed. The execution time of the function `Sun` based on C^2 was slower than that of JWS because JWS needed not download `SolarObject.class` and `200608.dat`, which were cached previously during the program start-up.

- After caching, C^2-based execution of each function was slightly faster than that of a single complete Jar because the C^2-based application could read the cached data resources without performance overheads of dynamic Jar unpackings. Furthermore, C^2-based execution of functions `Sun` and `Usage` that included cache coherence detections were still shorter than those of JWS. This means that, based on the tested network speed, the overheads of cache coherence detections were smaller than those of dynamic Jar unpackings.

- By comparing the elapsed times (before cached) of the function `Usage` between JWS and the single complete Jar, the total caching latency for `Help.jar` that was performed early in JWS could be estimated as 2910 milliseconds This latency would be the wasted time imposed by the early cache validation mechanism of JWS if the functions `Usage` and `Home location` were not executed,

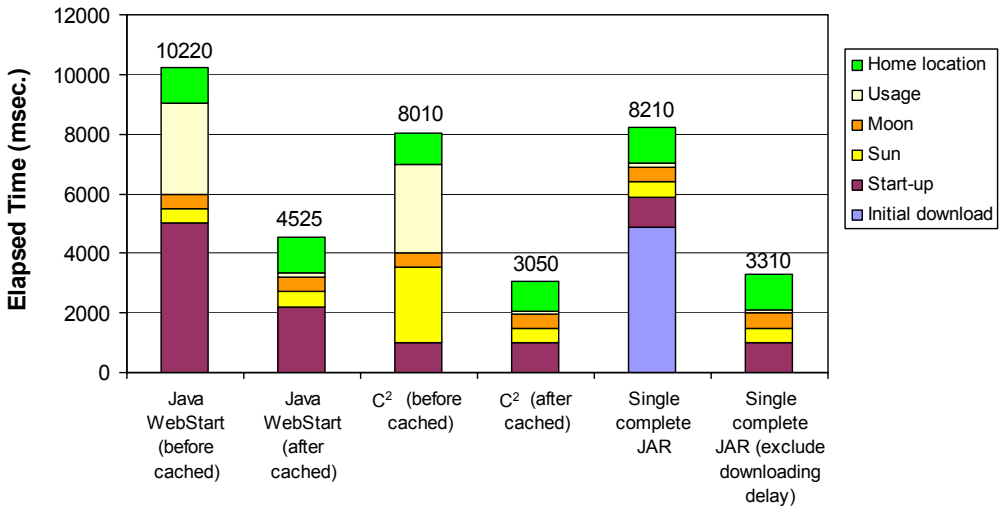


Figure 3.8: Total execution Times

and network bandwidth would also be wasted for transferring a 1-MB unused copy of `Help.jar`.

3.7.3 Overall Execution

Figure 3.8 portrays the total program execution times by combining the results from Figure 3.6 and Figure 3.7 together. Total execution time implied how much total performance overhead imposed by each deployment approach.

- Before caching, the total execution times based on C^2 were 200 milliseconds faster than those of the single complete Jar. This implied that the total cost of incremental cachings was lower than that of dynamically accessing data resources in Jar format, which involved dynamic Jar unpackings.
- After caching, the total execution time based on C^2 was 260 milliseconds faster than that of the single complete Jar. This speed-up, which was greater than that of before caching (200 milliseconds), indicated that the total cost of incremental cachings (existing in the case of before caching) was higher than that of incremental cache validations (existing in the case of after caching).
- Before caching, the total execution time based on C^2 was 2210 milliseconds shorter than that of JWS and, after caching, 1475 milliseconds faster than that

of JWS because of the same reasons: JWS's self-loading latency and dynamic Jar unpacking latencies. However, the former speed-up was larger than the former because the cost of early cache validation in the case of JWS after cache was smaller than the total cost of incremental validations in the case of C^2 after cache.

Chapter 4

Object Class Clustering Approach

4.1 Introduction

It is usual that only a subset of whole OO program classes is used by each program function. This fact opens up opportunity to optimize program performance and system resource usage. For example, a start-up process of an interactive program typically need no classes of the program's help function. In particular, optimization of program start-up time is a loading of only initial subset of entire program classes when the program is launched. (Loading of the help function classes can be done at later point of time on demand, i.e., incremental loadings of deferrable classes). Note that the term *load* refers to downloading over the network or loading from local storage. As another motivative example, exception handling classes might not be used in a program execution. By loading really needed classes on demand instead of eager loading whole program classes that potentially include unused ones, system resources (e.g., network bandwidth, computer memory, and CPU time) can be prevented from being wasted.

To achieve the program loading optimizations, an approach used to determine the deferrable classes is essential. Instead of loading deferrable classes one by one at each time, it is more efficient to load a *cluster* of deferrable classes at once to reduce overheads in computation, memory space, and network bandwidth. In other words, the unit of loading should be a cluster.

As a matter of fact, the idea of partial and incremental class loading has been gaining more interest and is enabled by several recent technologies of dynamic program deployment, such as lazy resource downloading of Java Web Start (34), Java dynamic class loading (42), and code splitting technique (14). Unfortunately, lacking

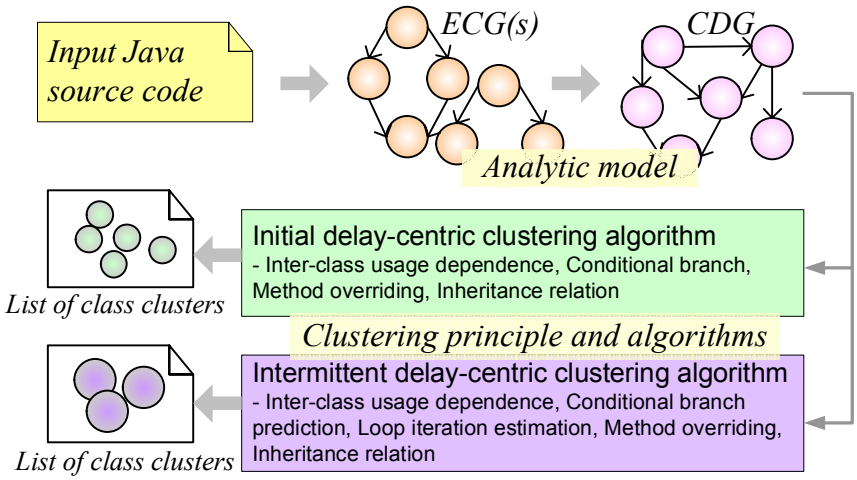


Figure 4.1: Overview of the class clustering approach

simple and systematic method for class clustering to complement such technologies is a major hindrance to the program loading optimization. This shortage becomes even more important to the large or complicated programs. To the authors' knowledge, very little effort has been made on the development of class clustering method for program loading optimization: (63) proposed a class clustering approach for object caching that is difficult to use because it involves not only execution profiling but also dynamic runtime analysis. Therefore, a new static approach for class clustering that is practically useful for users of above mentioned technologies including static tool developers was proposed.

4.2 Class Clustering Approach

The proposed approach performs clustering analysis on input Java program source codes. Figure 4.1 provides an overview of the clustering approach, which is detailed as follows.

4.2.1 Clustering Principle

The approach relies on a simple, intuitive principle that is "classes that are potentially used in the same time interval and address space should be assigned to the same

cluster to be loaded at once". This principle improves both *spatial locality*[1] and *temporal affinity*[2] that are crucial to the optimization of code loading. For the sake of conciseness, classes that exhibit both spatial locality and temporal affinity are referred to as they exhibit *contemporaneous proximity*, which is defined as a time-space property which indicates an ensemble of entities (classes) that are used in the same time interval and space. Classes that manifest contemporaneous proximity can benefit from both grouping and (implicit) prefetching, i.e., cluster (as a loading unit).

The approach exploits following criteria to identify Java classes that exhibit contemporaneous proximity. The criteria are explained based on Figure 4.2. Note that the term classes are used to refer to user-defined classes; Java system classes need not be considered.

- **Inheritance relation**: A class and its all ancestors (including implemented interfaces) must be available together to the program's class loader. Therefore, `Editor` must be agglomerated with `TextEditor`.

- **Inter-class usage dependence**: Usage dependence between two classes occurs when one class invokes any method of the other class. If the invocation occurs with high probability, both classes should be assigned to the same cluster. For example, `TextEditor` class definitely uses `Comp3`, both should be available in the same cluster. In fact, even though classes that do not invoke a method of each other but are always used together by some other class should also be assigned to the same cluster. For example, classes `Comp1` and `Comp2` belong to the same branch of a conditional branch statement, they are always used (or not used) together by `TextEditor`. Consequently, both `Comp1` and `Comp2` should be placed in the same cluster. The latter kind of cluster *loose cluster* is called as it contains loosely-coupling classes. In other words, a loose cluster incorporates classes that belong to the same basic block.[3]

- **Non-loop conditional branch** (or conditional branch in short): It is important to take into account a conditional branch construct since it decreases the

[1] Classes exhibit spatial locality(58) when the use of one class indicates future use to class in nearby memory space. These classes can benefit from "grouping".

[2] Classes manifest temporal affinity(53) when they are used in the same time period. These classes can benefit from "prefetching".

[3] In compiler optimization, the term "basic block" refers to a straight-line piece of code without any jump or jump target in the middle; jump target, if any, starts a block, and jump ends a block.

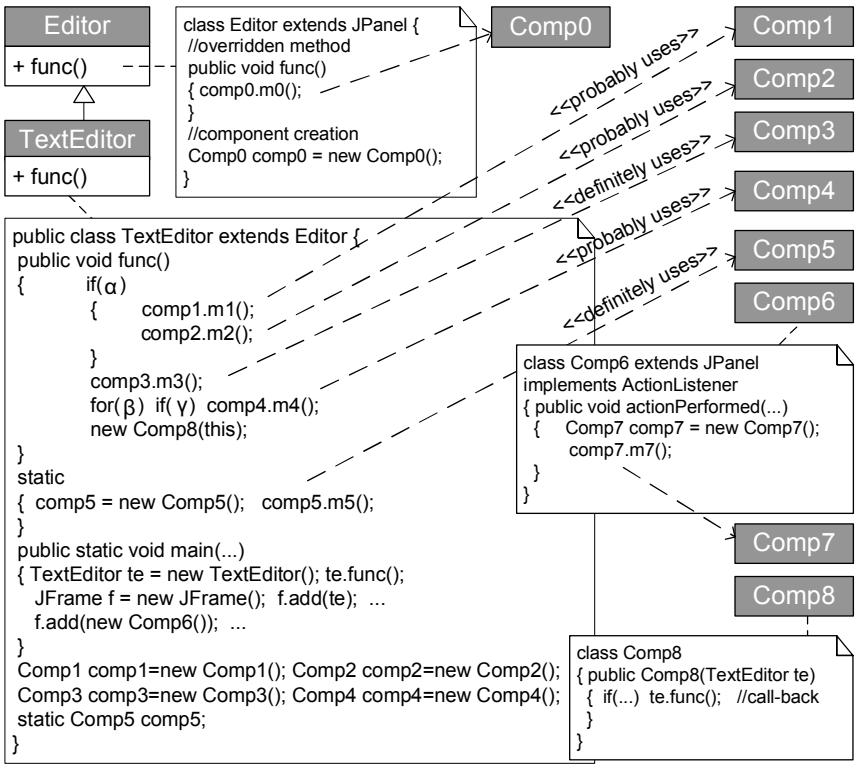


Figure 4.2: UML(50) class diagram of an example Java program that contains fundamental programming patterns

usage probability of classes. For example, if the `if(`α`)` statement that `Comp1` resides has low probability of being taken, `Comp1` (and `Comp2`) should not be assigned to the same cluster as `TextEditor`. Of course, class `Comp3` used outside the conditional branch should be agglomerated with `TextEditor`. (cf. Appendix C.1 for Java constructs that are regarded as non-loop conditional branches.)

- **Loop**: In a nested structure of both loop and conditional branch constructs, the iteration of loop increases the usage probability of classes inside unlike conditional branch. For example, whether or not class `Comp4` should be agglomerated with `TextEditor` depends on the number of iterations of `for(`β`)` statement and the probability of `if(`γ`)` statement being taken.

- **Method overriding**: Since `Editor.func()` method is overriden in `TextEditor` subclass, class `Comp0` will not be used by `TextEditor`. Thus, there is no need to

have `Comp0` in `TextEditor`'s cluster. This kind of situation often occurs when `Editor` is a reused component.

4.2.2 Analytic Model

In order to analyze an input Java source code against the above criteria, the approach engages a graph-theoretic model that consists of two kinds of graph, *Enhanced call graph* and *Class dependence graph*. The former is used as an intermediate representation to reveal potential method invocations of an input program. The latter aims to provide a unified view of all application classes and their inter-dependences based on the former graph to enable actual clustering analysis. The model particularly takes Java programming features into account to ensure the approach's practicality in Java context.

Enhanced call graph (ECG) enhances a conventional program call graph (28)—a directed rooted graph representing potential invocations among program methods—with control flow information. Hence, ECG consists of:

- *Method vertex* denotes a class method. A root vertex of ECG is always this kind of vertex.
- *Conditional branch vertex* is a weighted vertex that denotes a conditional branch construct. Each conditional branch vertex has a probability of corresponding conditional branch being taken as a vertex weight. (cf. Appendix C.2 for conditional branch prediction.)
- *Loop vertex* is a weighted vertex that denotes a (count-controlled or condition-controlled) loop construct. The vertex' weight represents the number of loop iterations. (cf. Appendix C.2 for loop iteration estimation.)
- *Edge* is a directed edge that denotes a transit of program control flow caused by method invocation, conditional branch, or loop.

The traditional program call graph, root vertex represent a program's entry method. A root vertex of ECG is used to represent `main()` method, Java Applet's `init()` method, or other system-invoked method (i.e., user-defined constructor, static initializer, `java.awt.event.*Listener.*()`, or user-defined system-invoked method).

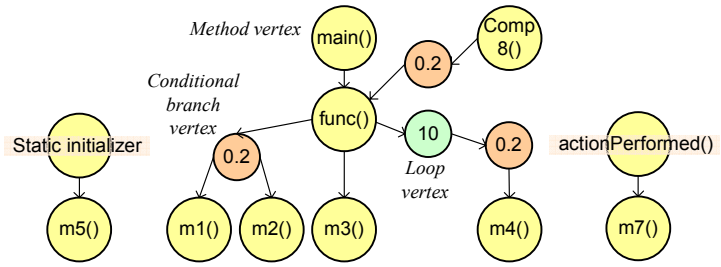


Figure 4.3: ECGs of the program in Figure 4.2

Note that the system-invoked methods of a program are not explicitly invoked at the source code level; and there can be multiple ECGs for each Java program.

Figure 4.3 demonstrates ECGs derived from the single program in Figure 4.2 based on two assumptions: (1) assume that `m1()`, `m2()`, `m3()`, `m4()`, `m5()`, and `m7()` do not use any other instance variable (which can be in form of field, local variable, or parameter), so they have no successive vertices; (2) assume that the probability of every `if()` statements being taken is 0.2, and the number of `for()` statement's iterations is 10. Note that by exploiting a call graph construction framework proposed in (28) (and their inter-procedural analysis tool (20)), the method overriding criterion previously described is automatically satisfied during the construction of ECG. Consequently, `Comp0` is suppressed from the ECGs in Figure 4.3.

Class dependence graph (CDG) is a directed rooted graph that represents a static view of dependences among classes of a program. It consists of:

- *Class vertex* represents a class.
- *Conditional branch vertex* has the same concept as that of ECG.
- *Loop vertex* has the same concept as that of ECG.
- *Normal edge* has the same concept as ECG edge, plus a control flow to the creation of an event listener[1] class.
- *Event edge* denotes a flow of Java event. The edge is directed from a class vertex representing an event listener to another class vertex. The edge is labeled with

[1]In Java event model, an event listener is a class whose ancestor or itself implements an interface or subinterface of `java.util.EventListener`.

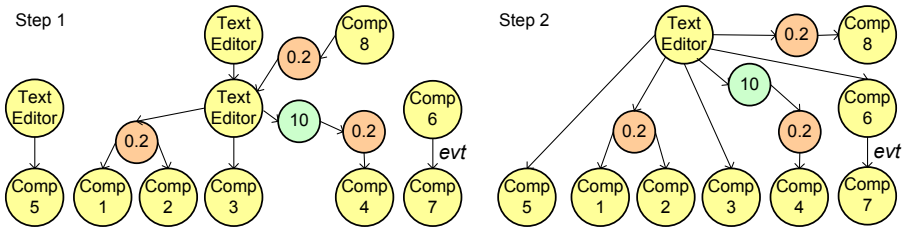


Figure 4.4: Intermediate CDGs (left) and final CDG (right) derived step by step from Figure 4.3

"`evt`". (It is important to model the event flow in CDG so as to analyze Java event-driven programs.)

Each program has a single unified CDG, which is used for clustering analysis. The CDG is constructed by transforming the ECGs in two steps as follows (Figure 4.4).
Step 1: *Reveal inter-class usage dependences* by substituting corresponding class vertices for all method vertices in every ECG. (Using the call graph construction framework of (28), each method vertex in ECG contains class information to be used in this step.) All edges derived from ECGs remain unchanged in CDGs except for edges directed from method vertices representing event listeners' standard event method (e.g., `actionPerformed()` vertices in Figure 4.3) that must be replaced with event edges. The result of this step is a set of intermediate CDGs (Figure 4.4 left).
Step 2: *Create a unified CDG* by the following steps:

(2.1) Considering intermediate CDGs as a whole, augment it with a normal edge between every pair of class vertices representing event listener creator and event listener. In Figure 4.4 (left), for example, class `Comp6` acts as an event listener because it implements `actionPerformed()`, while class `TextEditor` that instantiates `Comp6` is an event listener creator.

(2.2) All edges in a path that points toward a vertex denoting called-back[1] class (which is converted from a called-back method vertex of ECG) must be changed to reverse direction (e.g., the path between `TextEditor` and `Comp8` vertices in Figure 4.4).

(2.3) Merge all redundant class vertices (e.g., `TextEditor` vertices in Figure 4.4 left) into a single unique one. All normal edges incident to the merged vertices must

[1]Call-back technique is often used to implement event listeners in Java event-driven programs.

be retained. A final unified CDG (Figure 4.4 right) has a single root vertex.

4.2.3 Clustering algorithms

A CDG is taken as an input for cluster identification. Two variants of cluster identification process is provided as follows.

4.2.3.1 Initial delay-centric algorithm

This algorithm is conservative in the sense that if a class is solely used through conditional branch by another class, both classes will never be agglomerated together. The algorithm therefore ensures no wastes of system resources due to the penalty of conditional branch misprediction. This merit is significant to resource-constrained computing in which only classes that are absolutely used should be loaded.

The algorithm is presented by Figure 4.5 and described in details as follow.

- It traverses a CDG in depth-first order (just like the order used to create the ECGs from an input program). `rootVertexQueue` supports three operations: enqueue, dequeue, and remove a specified item (used on line 8).

- The inheritance relation criterion (Section 4.2.1) is determined through `superClassesOf()` and `implementedInterfacesOf()` on line 8 (based on the class information supplied by the call graph construction framework in (28) as aforementioned).

- The algorithm does not agglomerate event listener class and classes used by standard event method (called *event-driven used classes*) (line 14-15). A rationale is that the standard event method is never invoked by any application class but underlying system, thus the event-driven used classes are considered probably used by the event listener class.

- On line 17, if a new class vertex (as a root vertex of new traversal) belong to the same preceding conditional branch vertex as a starting vertex of a current cluster, the new traversal will assign any newly discovered member class to a current cluster (i.e., loose cluster) rather than to a new cluster.

algorithm `InitialDelayCentricClusterIdentification`
input CDG
output C: set of resulting clusters
declare $rootVertexQueue$: queue of root vertices of new candidate clusters
$Cluster$: currently identifying class cluster
v: visiting vertex
e: passing edge
P ←FALSE: flag indicating that there exists conditional branch vertex or event edge on the currently traversing path connecting adjacent class vertices; initial value of this flag is FALSE
cb: conditional branch vertex used to verify a loose cluster
begin
1: $v \leftarrow aRootVertexOf(CDG)$
2: **do**
3: $cb \leftarrow v.precedingConditionalBranch()$
4: Traverse CDG starting at v in depth-first order
5: **if** v is a class vertex
6: **if** P = FALSE
7: $Cluster \leftarrow v \cup superclassesOf(v) \cup implementedInterfacesOf(v) \cup Cluster$
8: Remove from $rootVertexQueue$ all occurences of v enqueued during identifying the current cluster
9: **else if** $v \notin Cluster$ //P = TRUE
10: $rootVertexQueue.enqueue(v)$
11: $P \leftarrow$ FALSE
12: **else if** v is a conditional branch vertex
13: $P \leftarrow$ TRUE
14: **if** e is an event edge
15: $P \leftarrow$ TRUE
16: $v \leftarrow rootVertexQueue.dequeue()$
17: **if** $v.precedingConditionalBranch() \neq cb$
18: $C \leftarrow Cluster \cup C$
19: $Cluster.clearMember()$
20: **while** $v \neq$ NULL
end

Figure 4.5: Initial delay-centric algorithm

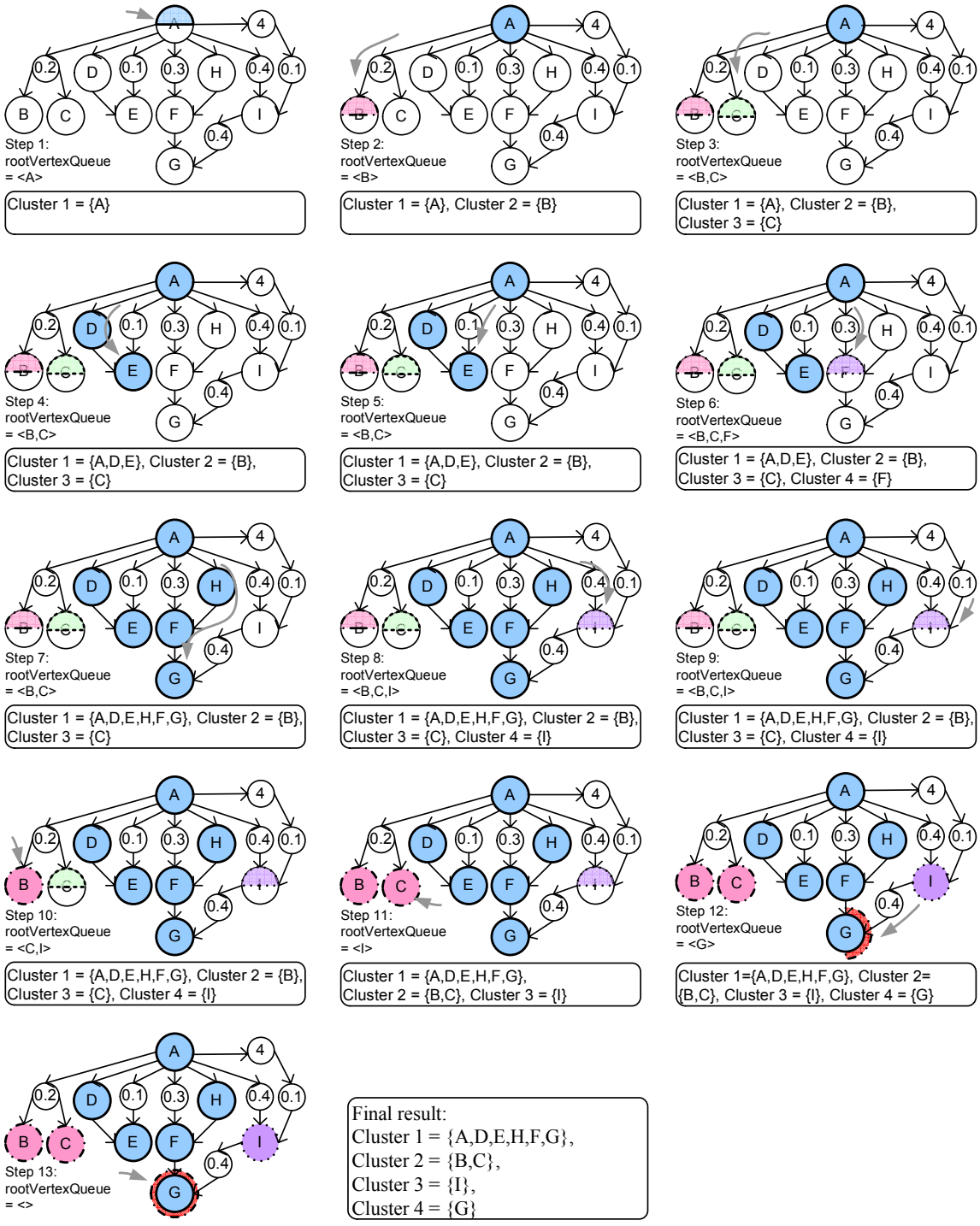


Figure 4.6: An example of initial delay-centric cluster identification

Figure 4.6 provides a comprehensive demonstration of this algorithm based on an input CDG. Of course, the input CDG to be used with this algorithm can be constructed inexpensively without the weight information of conditional branch and loop vertices because it is unnecessary. The details of important steps are as follows.

- Any vertex marked with half circle is a candidate for a new cluster's root vertex.
- In step 5, `E` is not a candidate for new cluster as it has been already agglomerated with Cluster 1.
- In step 7, `F`, which is regarded as a candidate for Cluster 4 by step 6, is instead included into Cluster 1 and no longer a candidate for Cluster 4.
- Step 11 detects a loose cluster, which makes `if()` statement on line 16 in Figure 4.6 evaluate to false because `C` belongs to the same preceding conditional branch vertex as `B`. As a result, `C` is agglomerated with `B` in a loose Cluster 2.
- Step 12 assigns `G` to a candidate Cluster 4 though `G` has already been a member of Cluster 1. In practice, the presences of `G` in both clusters do not mean that `G` has to be reloaded when loading Cluster 4 unless `G` in Cluster 1 (which has been loaded before Cluster 4) has become invalid (e.g., expired, or evicted from the locality of a client class loader). (Java Web Start, for example, follows this idea by looking locally before remotely for a valid class file.)
- Step 13 shows the final clustering result.

4.2.3.2 Intermittent delay-centric algorithm

This algorithm is speculative in the sense that it agglomerates a class and its all "potentially" used classes together. The potentially used classes are determined through a stochastic process, which consists of conditional branch prediction and loop iteration estimation. The algorithm trades off the ability to prevent system resources from being wasted (due to the penalty of both conditional branch misprediction and possibly loop iteration mis-estimation) for the chance of performance improvement (through aggressive prefetching of potentially used classes).

By extending the previous algorithm in Figure 4.5, the intermittent delay-centric algorithm proceeds as shown in Figure 4.7. As mentioned, the main difference from the pervious one of this algorithm is that it agglomerates all classes with another class

algorithm `IntermittentDelayCentricClusterIdentification`
input CDG
output C: set of resulting clusters
declare $rootVertexQueue$: queue of root vertices of new candidate clusters
$Cluster$: currently identifying class cluster
v: visiting vertex
e: passing edge
$pW \leftarrow$1.0: weight of currently traversing path connecting adjacent class vertices with initial value of 1
cb: conditional branch vertex used to verify a loose cluster
$totalpW$: total weight of all paths linking between root vertex of $Cluster$ and v
begin
1: $v \leftarrow aRootVertexOf(CDG)$
2: **do**
3: $cb \leftarrow v.precedingConditionalBranch()$
4: Traverse $ssCDG$ starting at v in depth-first order
5: **if** v is a class vertex
6: $totalpW \leftarrow pW$+Total weight of all paths to v that were found previously during identifying the current cluster
7: **if** $totalpW \geq 0.5$
8: $Cluster \leftarrow v \cup superclassesOf(v) \cup implementedInterfacesOf(v) \cup Cluster$
9: Remove from $rootVertexQueue$ all occurences of v enqueued during identifying the current cluster
10: **else if** $v \notin Cluster$ $//prevTotalpW < 0.5$
11: $rootVertexQueue.enqueue(v, pW, Cluster)$
12: $pW \leftarrow 1$
13: **else if** v is a conditional branch vertex $\vee$ v is a loop vertex
14: $pW \leftarrow pW * weightOf(v)$
15: **if** e is an event edge
16: $rootVertexQueue.enqueue(e.towardVertex())$
17: $v \leftarrow rootVertexQueue.dequeue()$
18: **if** $v.precedingConditionalBranch() \neq cb$
19: $C \leftarrow Cluster \cup C$
20: $Cluster.clearMember()$
21: **while** $v \neq$ NULL
end

Figure 4.7: Intermittent delay-centric algorithm

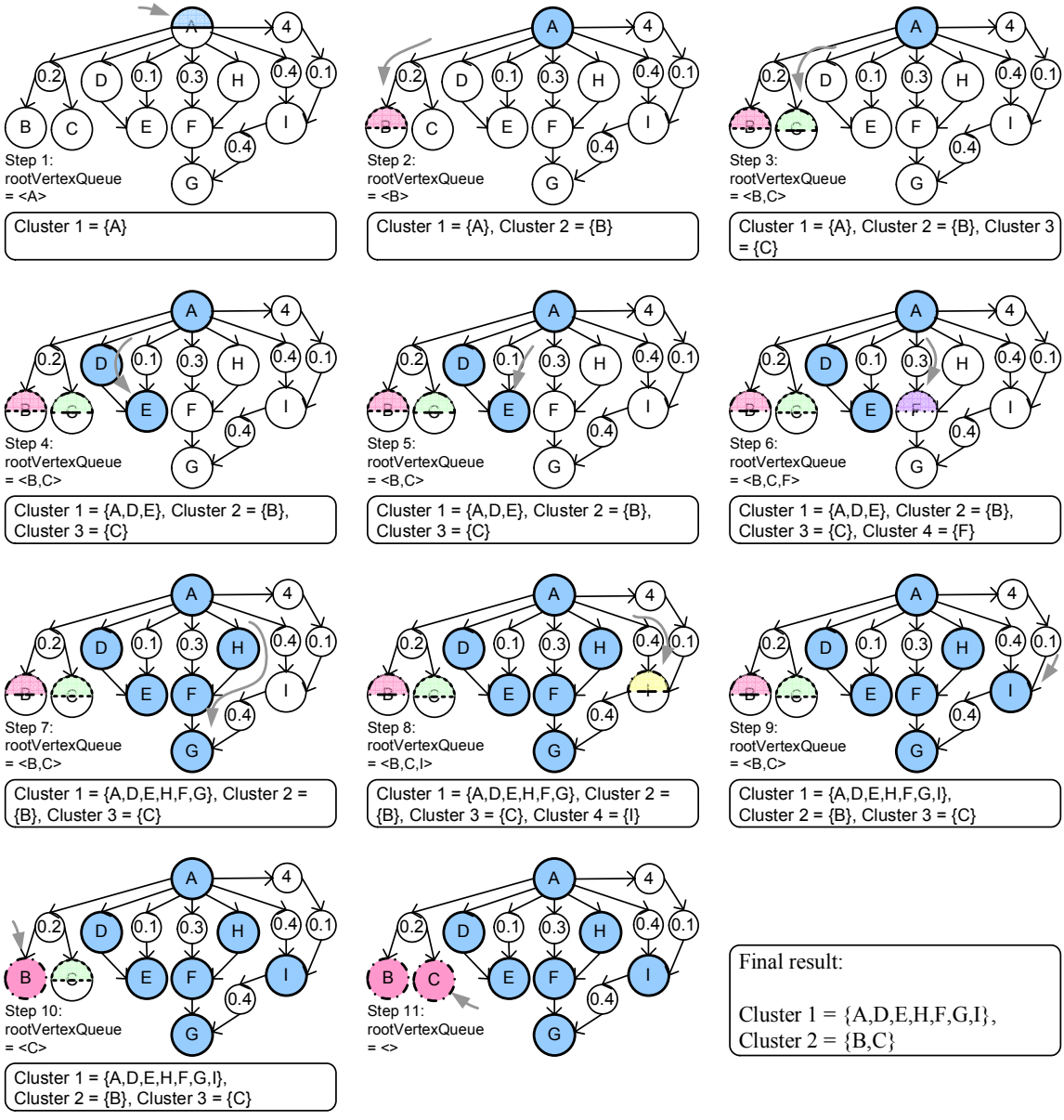


Figure 4.8: An example of intermittent delay-centric cluster identification

that uses them with probability at least 50%. However, event-driven used classes are not agglomerated with their corresponding event listener classes. To compute the previous total weight path with respect to a currently identifying cluster (line 6), a value of `pW` and a value of `Cluster` are associated with each vertex when enqueued into a `rootVertexQueue` (line 11). Method `enqueue()` on line 16 overloads `enqueue()` on line 11.

Figure 4.8 demonstrates a comprehensive cluster identification using this algorithm. In step 9, a total path weight to `I` becomes 4 multiplied by 0.1 plus the previous weight of the path to `I` computed in step 8 (0.4) that is totally 0.8. Step 11 detects a loose Cluster 2.

4.3 Algorithm Scalability and Effectiveness

The worst-case computational time complexity of the approach involves three major costs.

First, constructing a complete set of ECGs from an input program requires the construction of context-sensitive call graph, $O(N^{2 \cdot \alpha})$, where N and α are the number of call sites of a program and the average number of parameters per call site, respectively (28). Let CB and L be the number of conditional branches and loops in the program, respectively. The total construction cost of ECGs, represented with adjacency matrices, is $O((N^{\alpha+1} + CB + L)^2)$. In fact, ECG construction cost in the case of the initial delay-centric algorithm can be further optimized to $O((N^{\alpha+1} + CB)^2)$ in which actual cost to create each CB could be lessoned by omitting the branch prediction process (i.e., creating conditional branch vertices of ECG without vertex weights as aforementioned).

Second, to transform ECGs to a program-wide unified CDG consists of the following tasks: (1) Transform ECGs to intermediate CDGs, which costs $O(N)$, where N is the total number of vertices in all ECGs. (2) Add normal edges into the intermediate CDGs according to the number of event listener classes, R. (3) Redundant CDG vertices are determined and merged in $O((N^2 - N)/2)$ steps. Therefore, the cost of a unified CDG construction is $O(R + (N^2 + N)/2)$.

Finally, since both cluster identification algorithms fully traverse an input CDG, their costs are bounded to $O((C + CB + L)^2)$ where C, CB, and L are the number of class vertices, conditional branch vertices, and loop vertices, respectively.

Program	Description	Number of source code lines	CS	CB	L	Evt
JMail(66)	E-mail client	10305	2504	683	171	36
PlanetFinder(19)	Celestial map applet	2479	345	151	16	1
Webpad(35)	Sun's word processor	1164	276	74	6	13
JavaDoc(27)	API document generator	1320	314	138	27	0
JDepend(32)	Design quality measurer	3151	141	137	56	0

Table 4.1: General information and CDG quantitative features of the analyzed Java programs. (The number of source code lines excludes comments. Columns `CS`, `CB`, `L`, and `Evt` show the respective number of call sites, non-loop conditional-branch blocks, loop statements, and event-driven used classes in each program.)

Because the costs above are the order of quadratic in the sizes of their inputs, both algorithms are significantly scalable to any program size.

The effectivenesses of both algorithms were studied by applying them to real Java programs presented in Table 4.1. The study focused on interactive programs (which are also GUI-based and event-driven) since their start-up latency—the time period between the program's initial invocation and the entry of control flow into a main event loop to start responding to user activity—could be immediately perceived by user. Nevertheless, the study also used non-interactive programs, `JavaDoc` and `JDepend`, for analyses to show if the programs could gain advantages from class clustering.

The comparative results of cluster identifications are provided by Figure 4.9. From these results, the following conclusion could be drawn.

- The results varied across the tested programs even of the same kinds (i.e., interactive or non-interactive one). These results, however, substantiated the applicability of the clustering principle for Java programs.

- For loosely-coupled programs (in which most classes use or are used by others with low probability), such as `JMail`, `Webpad`, and `JavaDoc`, they could gain benefits from the clustering, especially `JMail` and `Webpad` that contained several event-driven used classes.

- The effects of both algorithms were not obvious in tightly-coupled programs, such as `Planet Finder` and `JDepend`, in that the results tended to consist of

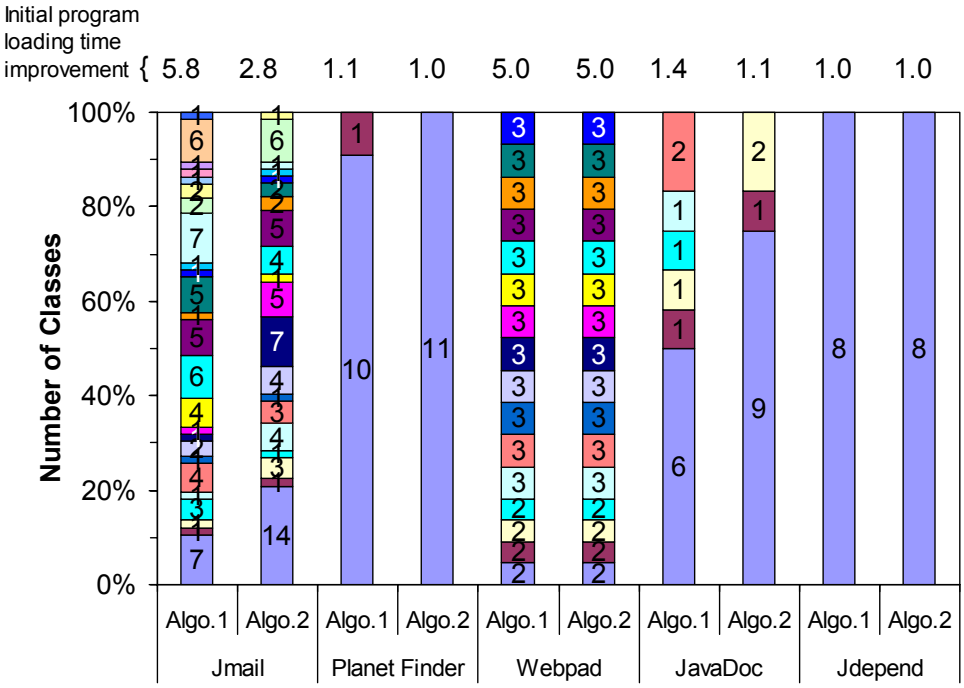


Figure 4.9: Clustering results and effectiveness of the clustering approach. (`Algo.1`: Initial delay-centric algorithm, `Algo.2`: Intermittent delay-centric algorithm; Each stack in graph bars denotes a cluster and labeled with the number of cluster member classes; The lower stacks were encountered by the algorithms earlier than the higher ones, the lowest stack is an initial cluster that contained a program's root class.)

a single large cluster and some small clusters (due to most classes of such programs exhibit a high degree of contemporaneous proximity). This algorithmic behavior is particularly plausible for two reasons. First, aggresively decomposing the large cluster into finer-grained clusters would cause frequent overheads of incremental loadings. Second, when some small clusters are not loaded, program response time could still be improved especially in unreliable network environment where downloading a small piece of code potentially imposes long delay.

- For both kinds of program, the initial delay-centric algorithm mostly produced finer grained set of clusters than the intermittent delay-centric one. This is an evidence of the algorithms' major characteristics (i.e., being conservative or speculative).

- The improvement of initial program loading latencies by the algorithms are shown on top of Figure 4.9. They were approximated from the total bytecode size of all clusters per the bytecode size of an initial cluster (which contained a program's root class). The approximation shows the effect of code transfer speeding up. Besides the code transfer time, actual program start-up latency also involves an initial program execution and file processing latencies required by an execution environment. Also note that each initial cluster was used as a minimum part for successful program execution. For interactive programs, their initial clusters were sufficient to make them enter the main loops of event.

  The initial program loading times based on the initial delay-centric algorithm were never worse than those of the intermittent delay-centric one of the same program as demonstrated by the experimental results.

- The algorithms enabled the chances to economize system resources for almost all tested programs except `JDepend` (as it could not be decomposed). Both algorithms resulted in similar effect of system resource waste prevention in `Webpad` and `JDepend`. However, how much the system resources could be economized in real program execution depends on real entire program usage behavior and, in fact, the precision in ECG construction. The amount of economized system resources could be estimated from the total bytecode size of non-loaded clusters of each program.

- The algorithmic pros and cons could be exemplified as follows. Let us consider the `JMail` program, when executing its initial cluster, user of intermittent delay-centric algorithm has to prefetch seven classes more than that of initial delay-centric one. These classes were predicted by the former algorithm to have high probability of being used. If the classes are not really used, they will waste system resources; otherwise, they will give extra performance improvement since the intermittent delay-centric algorithm's user does not experience delays in incremental loadings of these additional classes unlike the user of initial delay-centric algorithm.

  The intermittent delay-centric algorithm, on the one hand, could provide extra performance improvement over the initial delay-centric one by aggressive code prefetching. On the other hand, the intermittent delay-centric algorithm might impose system resource waste due to mis-prediction in analysis.

Although the initial program loading time improvement is a major advantage of partial and incremental program loading technique, it is noteworthy that an unfavorable effect of this technique is intermittent delays due to incremental loadings. A conventional whole-at-once program loading scheme, on the other hand, trades off initial program loading time reduction for smooth program execution because all program components are brought locally before program launching. However, the partial and incremental program loading scheme has another benefit in opening up a chance to optimize system resource consumption. For example, some program component (e.g., help system) might never be used by a program execution, thus need not be loaded.

4.4 Related work

First, the applicability of existent clustering techniques in various domains were extensively discussed toward partial and incremental program loading. Then the pertinent class clustering schemes were discussed.

Clustering for CPU cache performance (16) improves both spatial locality and temporal affinity through data structure reorganizing. It solely targets the structure of data but program control flow. Static clustering in OODB field (29) reduces the number of disk I/O operations by placing frequently co-accessed persistent objects in the same disk block and at the same time by balancing the number of clusters between disk blocks. It overlooks OO polymorphism concept and is directed by a constraint of disk block size. Software clustering in parallel computation (18) is used as units of distribution in order to minimize communications among parallel computing nodes. It improves spatial locality, whereas temporal affinity is not considered. Cluster analysis in data mining domain (31) works based on actual data of the objects being clustered rather than relation among them (the relation is represented as a program call graph). It is not suited for performance optimization, the focus of this dissertation. To summarize, clustering schemes in these domains are not optimized for the program loading.

As closely pertinent work, CASCADE (63) provides an approach to figure out each Java class bundle (i.e., class cluster) to improve CORBA object caching performance. The approach relies on a distance measurement of a weighted class graph, which is constructed from (flow-insensitive) static and profile-based information and

dynamically updated according to runtime class loading behavior. Although this technique seems to be sophisticated, to use it is inconvenient (due to requiring the profiling information) and intricate (as pointed out on page 62 of the literature per se). Furthermore, this approach lacks reliable cluster identification technique because it simply uses a user-defined threshold to decide the boundary of each code bundle based on the expensively-constructed class graph. A slight change on such a threshold can lead to substantial impact on both performance and system resources.

The authors previously proposed a premature version of initial delay-centric algorithm (9). The algorithm generates CDGs from method-attribute dependence graphs. Unreachable classes due to method overriding are not excluded from CDGs, thus complicating cluster identification process.

Chapter 5

Conclusion

The philosophy of this study lies in the realm of fine-grained replications in distributed object computing. Fine-grained replication enables a remote object-oriented application to be replicated partially and on demand incrementally in locality.

5.1 Research Outcomes

The applicability patterns of the fine-grained replications in distributed object systems have been identified: object-cluster replication and class-cluster replication. The research scope has been drawn to focus on object-cluster middleware, class-cluster middleware, and class clustering method. The problems in related work have been researched and summarized in general along with the solutions as follows.

1. Lack of replication middlewares for pervasive client-server CSCW. SOOM object-cluster replication middleware (4; 5; 6; 8) has been proposed to resolve this problem. The architecture of SOOM is optimized into two tiers communicating with each other in a pull style. SOOM's server tier is responsible for (1) creating and returning the replicas in response to replication requests from clients, (2) updating the master copy of a servant application based on the received update messages, (3) maintaining inter-cluster reachabilities, (4) instantiating a consistency protocol and maintaining the centralized lock variables for concurrent access control, and (5) serving RMIs in a coordinated manner with the corresponding replica accesses. A client tier is responsible for (1) issuing replication requests and rebuilding the replicas from the responses, (2) recording

the updates and writing them back to the server, and (3) acquiring and releasing locks from the server to achieve the concurrency control and coexistence between RMI and fine-grained replication.

2. Lack of easy-to-use and efficient middlewares for Java application deployment over the mobile Internet. C^2, a client-side class-cluster replication middleware, has been devised to address this problem. C^2 (5; 10) provides a simple API by which a Java application program, including its actively used classes, can be partially and on-demand incrementally downloaded from a HTTP server. Cache validation of C^2 also operates on demand. C^2 is lightweight and conceals heterogeneity by using Java and HTTP, thus relatively portable on the Internet.

3. Lack of static clustering approaches for program loading optimization. Therefore, the initial delay- and intermittent delay-centric class clustering algorithms (7; 9) have been devised. The algorithms are specialized for Java programs to be practically useful. The algorithms guide the clustering based on the novel clustering principle called contemporaneous proximity, which is programming language independent. The algorithm-based optimized programs can be deployed by not only C^2 but also other equivalent technologies that support partial and on-demand incremental program loading.

Both SOOM and C^2 have been released publicly at http://research.nii.ac.jp/H2O/soom/index.html.

5.2 Empirical Results and Findings

The quantitative properties of SOOM were measured through the following empirical evaluations. First, experiments in single user and multi-user environments using different consistency protocols indicated the practical throughputs of SOOM-based application. Second, for the tested cluster, SOOM-based replication was not expensive compared to Java RMI as each member of the cluster must be accessed locally only three times to outweigh the costs of replication and update committing. Third, an experiment using the varied numbers of client processors assured the scalability of SOOM. Finally, an experiment on the memory space requirement of a cooperative application showed that SOOM could reduce significant amount of network bandwidth and client memory consumptions that is incurred when using the traditional

replication approach. Based on a realistic SOOM-based cooperative application and the real-world usage scenarios, the experimental findings were (1) replication at cluster granularity can yield better performance than RMI, (2) maintaining consistency at the cluster unit is practical as substantiated by the system throughput results, (3) the combinative deployment of fine-grained replication and RMI improves system performance against pure using RMI, and (4) fine-grained replication is suitable for an application that need not be shared in its entirety.

Empirical results showed that C^2-based application's launching latency was reduced by 83% of that of whole-at-once program deployment. Total program deployment and execution overhead based on C^2 was 22% less than that of a well-known Java Web Start. The experimental findings were (1) C^2 can be completely transparent to end users because distributing it together with applications yielded practical downloading latency and (2) the whole-at-once program deployment in Jar format can lead to longer total execution time than the C^2-based program deployment.

Applying the class clustering algorithms to real Java programs resulted in the significant improvement of initial program loading time. In specific, among the experimented Java programs, using the initial delay-centric algorithm and the intermittent delay-centric algorithm improved initial program loadings, on average, by 2.9 and 2.2 times faster than the whole-at-once program loading, respectively. The intermittent delay-centric algorithm could reduce the number of intermittent delays to half of the initial delay-centric algorithm. Experimental results also indicated that the algorithms are practically useful to not only interactive programs but also non-interactive programs. An experimental finding was that both algorithms open up the chances to economize on system resources for loosely-coupled programs.

As a general lesson learnt, fine-grained replication should be a supplement to the traditional means of replication rather than to replace it.

5.3 Contributions

In the context of Java-based object-cluster replication middleware, SOOM provides the following contributions: (1) the design and implementation of fine-grained replication and RMI-supporting middleware, which has the following novelties: (1.1) a single proxy-based cluster realization framework, (1.2) a cluster table-based cluster validation technique, (1.3) a cluster table-based cluster loss prevention technique,

(1.4) a single-step cluster creation update committing technique, (1.5) an efficient inside-client consistency maintenance approach, (1.6) a quality-of-service manageable non-blocking consistency API, (1.7) a proxy and cluster table-based technique for centralized maintenance of inter-cluster reachabilities, (1.8) a relaxed version of Entry consistency protocol called Exclusive-write, which offers better efficiency than the Entry consistency, and (2) the performance analyses of combinative deployment of fine-grained replication and RMI.

Compared to related Java-based technologies for class-cluster replication, C^2 provides finer-grained replicability of actively used classes at the expense of simple program modification, transparently on-demand incremental updatability, and simpler data resource accessability.

The proposed class clustering principle "contemporaneous proximity" and the initial delay-centric and intermittent delay-centric clustering algorithms are original from the viewpoint of static program analysis.

5.4 Limitations and Premises

SOOM is optimized for pervasive client-server systems through the two-tier architecture and pull communication model. Thus SOOM is unable to attain object-cluster replications in peer-to-peer applications. Also, SOOM has made three assumptions. First, a cluster replica is accessed frequently and sufficiently to outweigh the cost of replication and update committing. This assumption makes object-cluster replication cost effective because total latency of replication-based invocations becomes smaller than that of remote method invocations in equal number. Second, most member objects in a cluster replica are modified before the replica is written back to the server. It would not be efficient if on few member objects of the cluster were modified but the whole cluster had to be committed to the server. Whether this assumption is valid or not also depends on the effectiveness of an applied object clustering method. Last, nested critical sections, if there are any in a client program, do not occur on the consecutive clusters holding mutual references. This is to prevent deadlock, for example, when each of two different clients currently locking two different clusters is trying to lock the other cluster locked by the other client can lead to deadlock depending on an applied consistency semantics and involved lock types (write or read).

Although C^2 seems to have several advantages, they come with the expense of program modification to utilize C^2's API. This also prevents the ease-of-development when the program codes using the API are often modified as the programs must be often re-compiled as a consequence. C^2 has made an assumption: all class files and data resources in a downloaded class cluster (Jar file) are utilized before they are replaced with the up-to-date ones. Otherwise, there is no benefit to do class clustering. The assumption's validity is also influenced by a used class clustering method.

The proposed class clustering algorithms turned out to be not useful for tightly-coupled programs as substantiated by the experimental results.

5.5 Recommendations for Future Researches

The first research direction on middlewares is drawn towards mobile computing. An emphasis should be made on how to fit them into the small memory spaces of mobile computing devices without losing the major functionality. Porting SOOM and C^2 into mobile computing platforms, such as Java ME platform (36), more or less incurs the tasks of re-design due to the limited set of platform library.

Second, evolving SOOM and C^2 into the complete caching systems to reduce client's memory space or persistent cache space requirements needs further research on fine-grained cache replacement.

Last, particular research on SOOM is recommended to improve the reliability of SOOM-based applications. Because SOOM's consistency protocols rely on the shared locks, if a client holding a lock crashes, other clients that desire the lock will have to wait forever. One solution is applying a lease concept (21) to control the lifetimes of locks to ensure that no client is kept waiting forever due to associated lock is never released. To bear in mind, using the leases degrades overall system performance, further experiments can be conducted to observe the performance degradation. In terms of API, lease-relevant parameters (e.g., lease duration) should be parameterized through `CRB.init()` method. Initializing a `CRB` without supplying the lease-relevant parameters can lead to using a default mode in which no lease is applied (i.e., unreliable operation).

With respect to a clustering approach, two points of improvement can be performed on the proposed class clustering algorithms. First, since the static analysis by

its nature cannot accurately cope with classed that are dynamically used via Java reflection mechanism, a solution to address this issue is by using a program annotation technique to enable programmers to supply their application knowledge as the hints of the potential classes that are dynamically used. Second, the program annotation can also be used to improve the precision of conditional branch and loop iteration analyses and to hint the prediction of event flow that is useful for the agglomeration of event listener classes and event-driven used classes based on the evaluation of event flow probability.

Future researches also include an object clustering algorithm based on the contemporaneous proximity principle. Alternatively, object re-clustering in a dynamic manner based on the real access behaviors should be researched. This can be achieved by means of runtime instrumentation.

There is still a long way to go on the research of fine-grained replication. The last scene of the story is hoped to be there where the fine-grained replication is as commonly deployed as the coarse-grained one.

Appendix A

SOOM's Design Structure

SOOM package consists of four subpackages. They are recursively presented as UML class diagrams as follows.

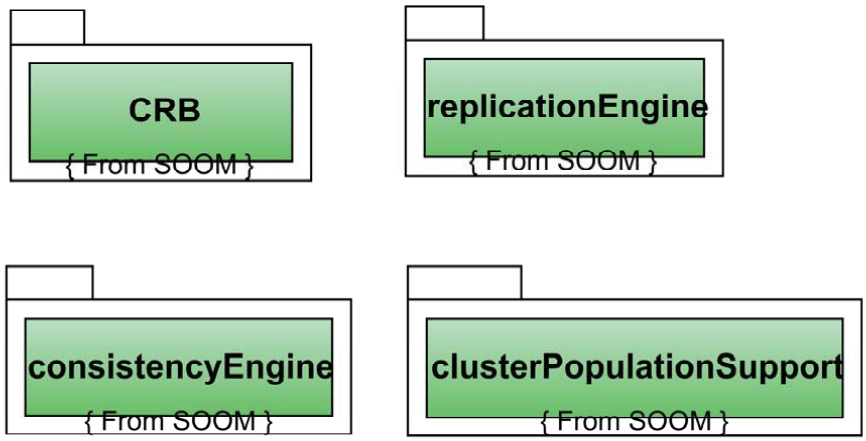


Figure A.1: SOOM package

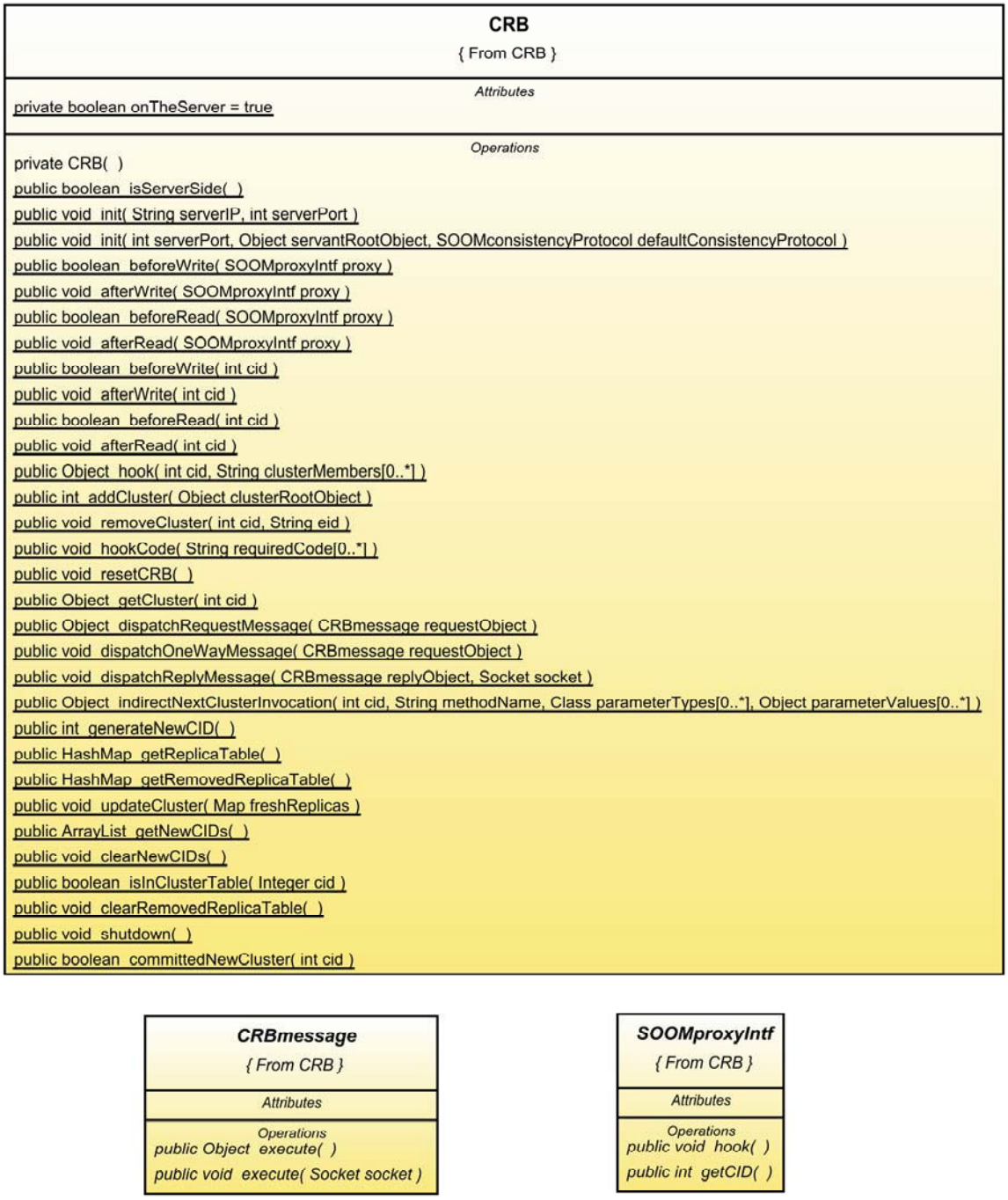


Figure A.2: SOOM's CRB subpackage

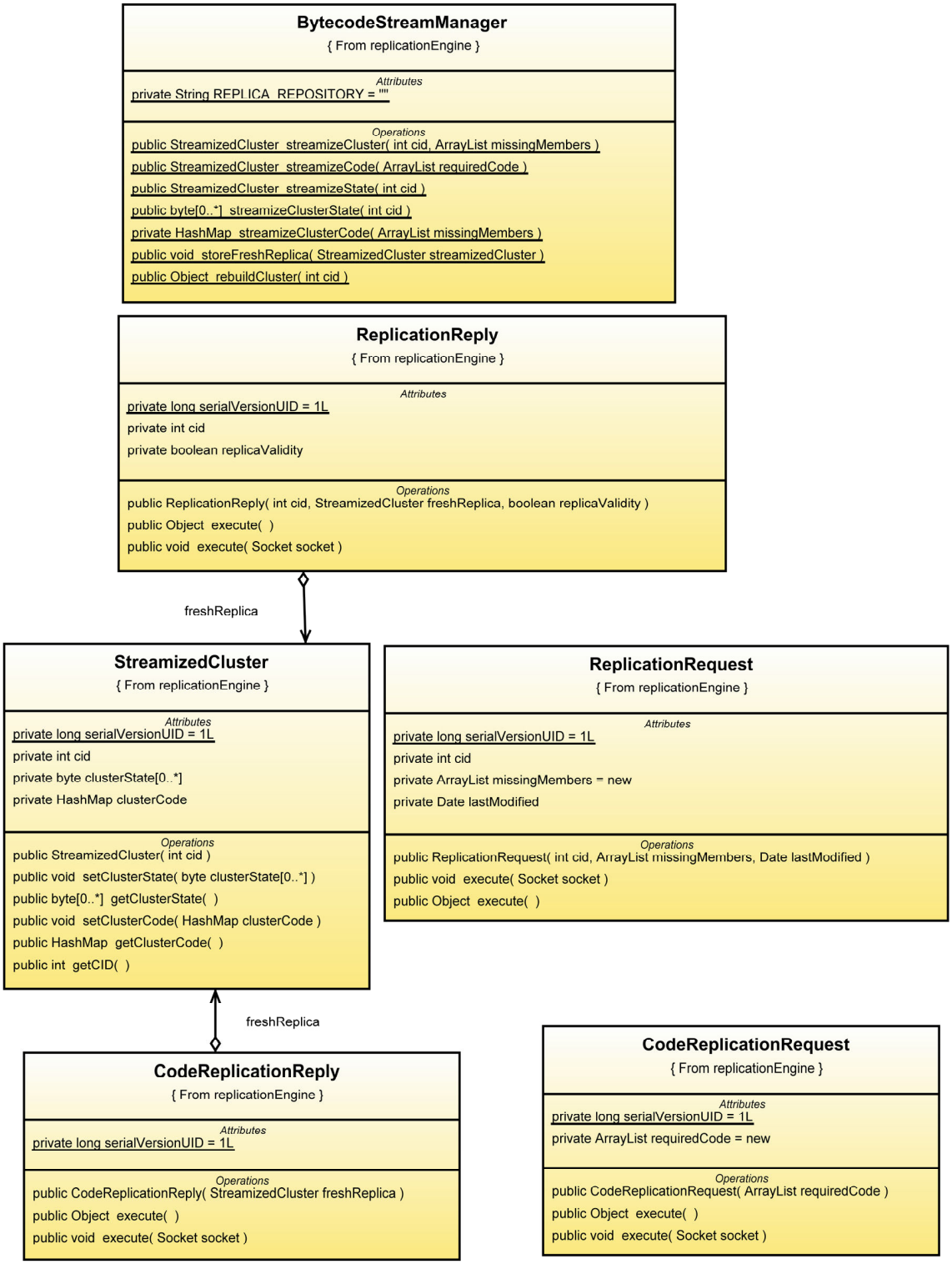


Figure A.3: Replication engine subpackage

Figure A.4: Consistency engine subpackage

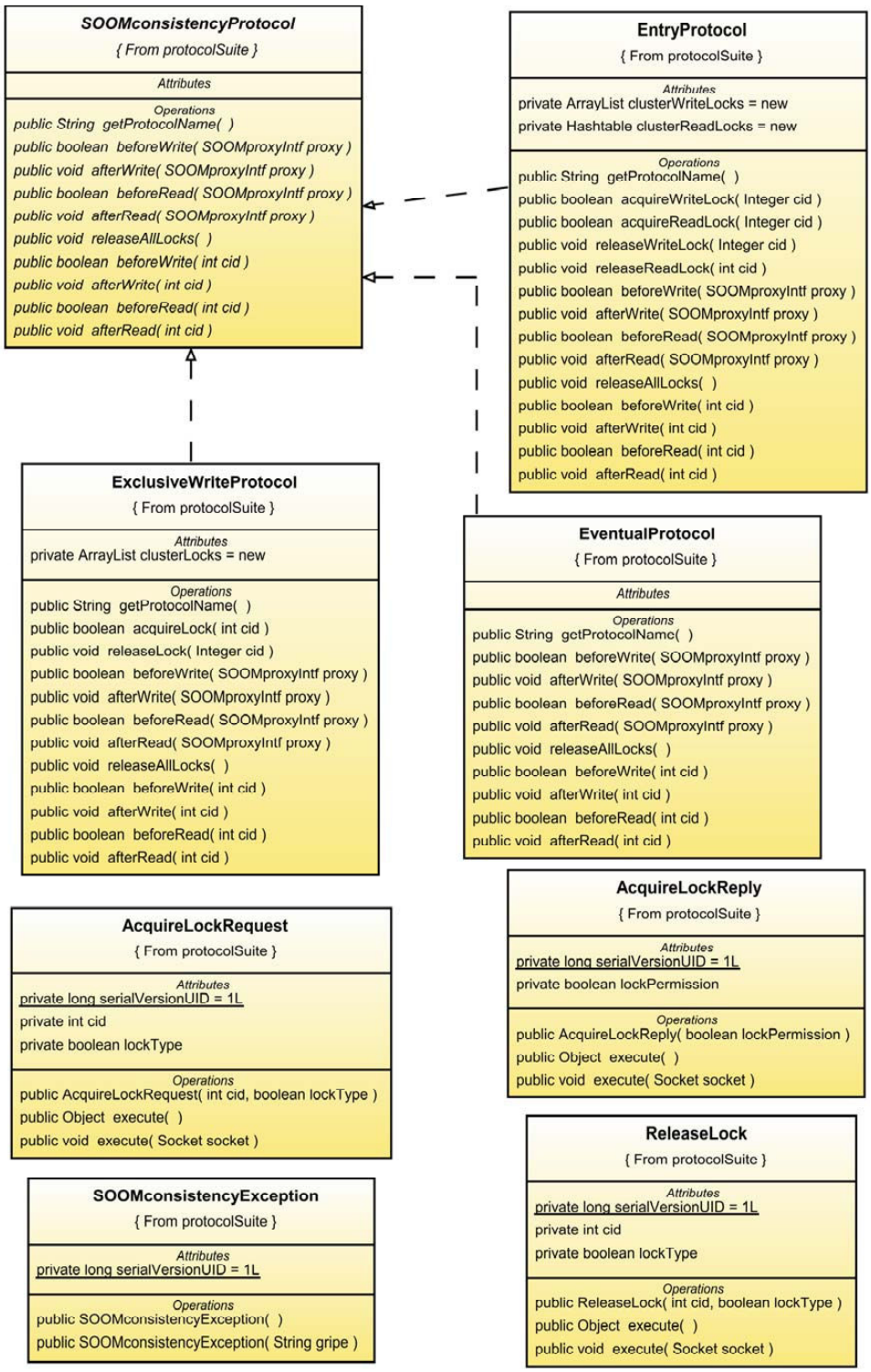


Figure A.5: Consistency protocol suite subpackage

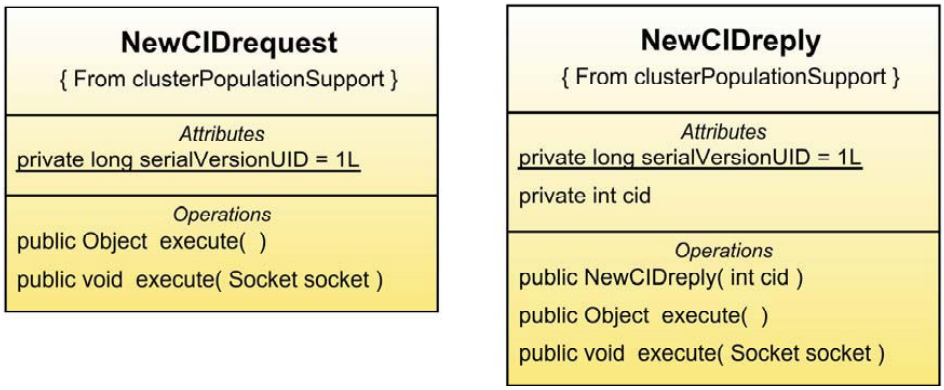


Figure A.6: Cluster population subpackage

Appendix B

C^2's Design Structure

C^2 package consists of a single self-contained class. It is recursively presented through the following UML class diagrams.

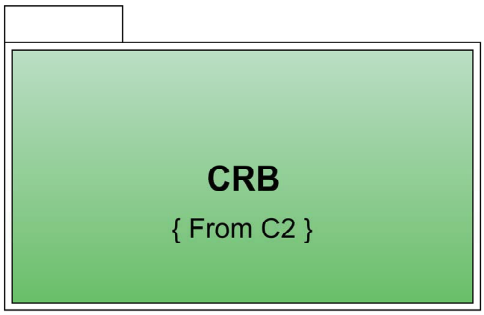


Figure B.1: C² package

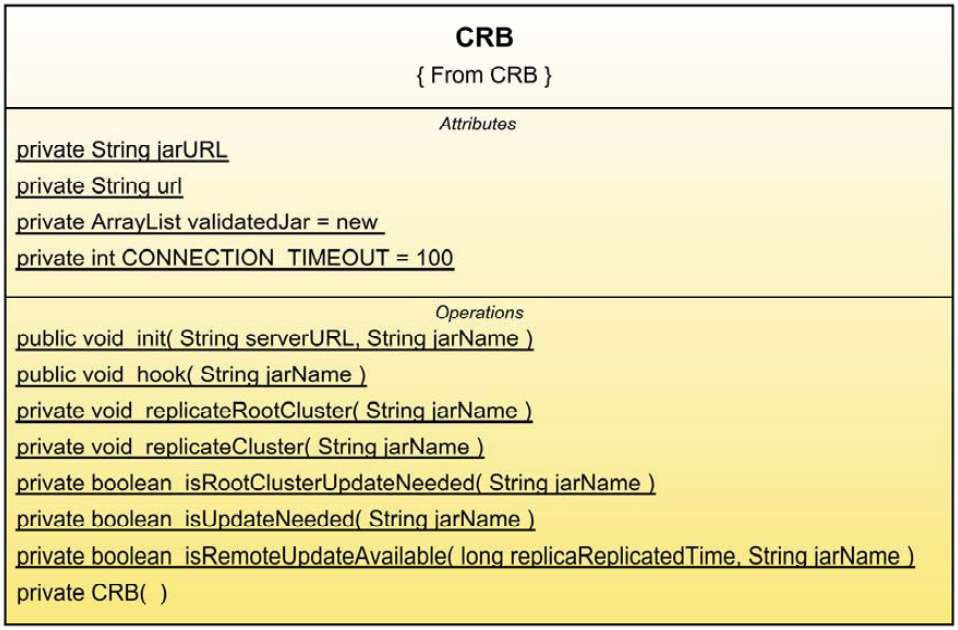


Figure B.2: C²'s CRB subpackage

Appendix C

Class Clustering Algorithm

C.1 Non-Loop Conditional-Branch Equivalent Constructs in Java

- `if`, `if-else`, and `if-else-if` statements.
- `switch` statement: The number of `case` blocks including `default` blocks are regarded as `N` in an N-way conditional branch.
- `catch` and `finally` blocks in `try-catch` statement are equivalent to `if` statement and its variants.

C.2 Conditional Branch Prediction and Loop Iteration Estimation

The probability of conditional branch being taken is predicted by exploiting the following techniques in fallback order:

1. Value Range Propagation (52): When a conditional branch is nested inside a count-controlled loop whose iteration range is determinate at compile time, this technique is applied.
2. Two-way conditional branch prediction heuristics (3): They are applied in the following order: *Call*, *Return*, *Loop*, and *Guard*. A branch that is predicted is

given probability of being taken equal to 74% (according to an average misprediction rate reported in (3)).

3. Naive strategy: The probability of each branch being taken of an N-way conditional branch is calculated by 1/N (where N $\geq$ 2, i.e., *target* and *fall-thru* branches).

The number of loop iterations is estimated by exploiting the following techniques in fallback order:

1. If a constant is explicitly used to carry a loop, such a constant can be directly used as a loop vertex' weight.

2. Constant propagation (64): When a loop-control variable is constant on all possible executions of a program, this technique is used to discover an actual value of such a variable.

3. A heuristic value of 8.

Bibliography

[1] Y. Aridor, M. Factor, A. Teperman, T. Eilam, and A. Schuster, *A high performance cluster jvm presenting a pure single system image*, JAVA '00: Proceedings of the ACM 2000 conference on Java Grande (New York, NY, USA), ACM Press, 2000, pp. 168–177. 43

[2] R. Bakalova, A. Chow, C. Fricano, P. Jain, N. Kodali, D. Poirier, S. Sankaran, and D. Shupp, *Websphere dynamic cache: Improving j2ee application performance*, IBM Systems Journal **43** (2004), no. 2. 2, 39

[3] T. Ball and J.R. Larus, *Branch prediction for free*, PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation (New York, NY, USA), ACM Press, 1993, pp. 300–313. 99, 100

[4] T. Banditwattanawong, S. Hidaka, H. Washizaki, and K. Maruyama, *Proxy-and-hook: A java-based distributed object caching framework*, IEEE International Conference on Industrial Informatics, 2005. 83

[5] ———, *Cluster replication for distributed-java-object caching*, IEICE Transactions on Information and Systems **E89-D** (2006), no. 11, 2712–2723. 15, 41, 43, 47, 48, 83, 84

[6] ———, *Fine-grained replication for private-workspace and memory-constrained computings*, NPC 2006: IFIP International Conference on Network and Parallel Computing, 2006, pp. 84–92. 83

[7] ———, *Optimization of program loading by object class clustering*, IEEJ Transactions on Electrical and Electronic Engineering (2006). 84

[8] ———, *Soom: Scalable object-oriented middleware for cooperative and pervasive computings*, IEICE Transactions on Communications **E90-B** (2007), no. 4. 83

[9] T. Banditwattanawong, K. Maruyama, S. Hidaka, and H. Washizaki, *Contemporaneity-conscious clustering algorithm for distributed object caching*, PDPTA (H.R. Arabnia, ed.), CSREA Press, 2005, pp. 277–283. 81, 84

[10] T. Banditwattanawong, H. Washizaki, S. Hidaka, and K. Maruyama, *Partial and on-demand incremental deployment of java application program over the internet*, ISCIT '06: International Symposium on Communications and Information Technologies, 2006. 84

[11] Alex Berson, *Client/server architecture (2nd ed.)*, McGraw-Hill, Inc., New York, NY, USA, 1996. 6

[12] J. Bortvedt, *Functional specification for object caching service for java (ocs4j), 2.0*, JSR 107 (2001). 2, 6, 39

[13] F. Boyer and D. Hagimont, *A configurable rmi mechanism for sharing distributed java objects*, IEEE Internet Computing, vol. 5, 2001, pp. 36–43. 40

[14] B. Calder, C. Krintz, and U. Holzle, *Reducing transfer delay using java class file splitting and prefetching*, OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (NY, USA), ACM Press, 1999, pp. 276–291. 7, 47, 63

[15] CanyonBlue and Inc., *Collaborative uml development*, White paper (2001), http://www.canyonblue.com/. 13

[16] T.M. Chilimbi, M.D. Hill, and J.R. Larus, *Cache-conscious structure layout*, PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation (New York, NY, USA), ACM Press, 1999, pp. 1–12. 80

[17] G. V. Chockler, D. Dolev, R. Friedman, and R. Vitenberg, *Implementing a caching service a distributed cobra objects*, Middleware '00: IFIP/ACM International Conference on Distributed systems platforms (Secaucus, NJ, USA), Springer-Verlag New York, Inc., 2000, pp. 1–23. 2, 6, 39

[18] G. Craig, U. Bellur, and K. Shank, *Clusters: A pragmatic approach towards supporting a fine-grained active object model in distributed systems*, Proceedings

of the 9th International Conference on Systems Engineering, 1993, pp. 245–250. 80

[19] B. Crowell, *Planet finder (source code)*, (2005), http://www.lightandmatter.com/planetfinder/. 77

[20] J. Dean, G. Defouw, D. Grove, V. Litvinov, and C. Chambers, *Vortex: an optimizing compiler for object-oriented languages*, OOPSLA '96: Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (New York, NY, USA), ACM Press, 1996, (Downloadable from http://www.cs.washington.edu/research/projects/cecil/www/vortex.html), pp. 83–100. 68

[21] V. Duvvuri, P. Shenoy, and R. Tewari, *Adaptive leases: A strong consistency mechanism for the world wide web*, IEEE Transactions on Knowledge and Data Engineering **15** (2003), no. 5, 1266–1276. 87

[22] J. Eberhard and A. Tripathi, *Efficient object caching for distributed java rmi applications*, Middleware '01: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg (London, UK), Springer-Verlag, 2001, pp. 15–35. 6, 40

[23] Wolfgang Emmerich, *Software engineering and middleware: a roadmap*, ICSE '00: Proceedings of the Conference on The Future of Software Engineering (New York, NY, USA), ACM Press, 2000, pp. 117–129. 6

[24] N. Erdogan, Y.E. Selcuk, and O. Sahingoz, *A distributed execution environment for shared java objects*, Information and Software Technology, Elsevier **46** (2004), no. 7, 445–455. 43

[25] P. Ferreira, L. Veiga, and C. Ribeiro, *Obiwan: Design and implementation of a middleware platform*, IEEE Trans. Parallel Distrib. Syst **14** (2003), no. 11, 1086–1099. 6, 42

[26] Apache Software Foundation, *Java caching system*, (2007), http://jakarta.apache.org/jcs/JCSandJCACHE.html. 6, 39

[27] The Apache Software Foundation, *Javadoc (source code) version 1.6.5*, (2005), http://ant.apache.org/srcdownload.cgi. 77

[28] D. Grove and C. Chambers, *A framework for call graph construction algorithms*, ACM Trans. Program. Lang. Syst. **23** (2001), no. 6, 685–746. 67, 68, 69, 70, 76

[29] S. Guinepain and L. Gruenwald, *Research issues in automatic database clustering*, SIGMOD Rec. **34** (2005), no. 1, 33–38. 80

[30] D. Hagimont and D. Louvegnies, *Javanaise: distributed shared objects for Internet cooperative applications*, Middleware'98, 1998, pp. 339–354. 6, 40

[31] J. Han and M. Kamber, *Data mining: Concepts and techniques*, Morgan Kaufmann, USA, 2001. 80

[32] Clarkware Consulting Inc., *Jdepend (source code) version 2.9*, (2005), http://www.clarkware.com/software/JDepend.html. 77

[33] Sun Microsystems Inc., *Java^{TM} archive (jar) files.*, White paper (2004), http://java.sun.com/j2se/1.5.0/docs/guide/jar/. 7, 45

[34] ———, *Java web start.*, (2005), http://java.sun.com/products/javawebstart. 7, 46, 63

[35] ———, *Webpad (source code) version 1*, (2005), http://java.sun.com/j2se/1.5.0/. 77

[36] ———, *Java micro edition*, (2006), http://java.sun.com/javame/index.jsp. 87

[37] ———, *Reflection api specification*, (2006), http://java.sun.com/j2se/1.5.0/docs/guide/reflection/. 24

[38] ISO, *Information technology — open distributed processing — reference model: Overview*, International Standard ISO/IEC 10746-1 First edition (1998-12-15). 6, 15

[39] JBoss Inc., *Treecacheaop - an object-oriented cache*, User documentation, release 1.2.4 (2005). 40

[40] P. Jogalekar and M. Woodside, *Evaluating the scalability of distributed systems*, IEEE Trans. Parallel Distrib. Syst **11** (2000), no. 6, 589–603. 36

[41] R. Kordale, M. Ahamad, and M.V. Devarakonda, *Object caching in a corba compliant system.*, COOTS, USENIX, 1996, pp. 65–82. 2, 6, 39

[42] S. Liang and G. Bracha, *Dynamic class loading in the java virtual machine*, OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (New York, NY, USA), ACM Press, 1998, pp. 36–44. 7, 47, 63

[43] T. Lindholm and F. Yellin, *The java^{TM} virtual machine specification.*, Addison-Wesley, USA, 1999. 47

[44] B. Liskov, M. Castro, L. Shrira, and A. Adya, *Providing persistent objects in distributed systems*, ECOOP '99: Proceedings of the 13th European Conference on Object-Oriented Programming (London, UK), Springer-Verlag, 1999, pp. 230–257. 40

[45] J. Maassen, T. Kielmann, and H. E. Bal, *Parallel application experience with replicated method invocation*, Concurrency and Computation: Practice and Experience **13** (2001), no. 8-9, 681–712. 6, 41

[46] J. Maassen, R.V. Nieuwpoort, R. Veldema, H.E. Bal, and A. Plaat, *An efficient implementation of java's remote method invocation*, SIGPLAN Not **34** (1999), no. 8, 173–182. 2, 6, 39

[47] M.W. MacBeth, K.A. McGuigan, and P.J. Hatcher, *Executing java threads in parallel in a distributed-memory environment*, CASCON '98: Proceedings of the 1998 conference of the Centre for Advanced Studies on Collaborative research, IBM Press, 1998, p. 16. 43

[48] National Institute of Standards and Technology, *Nist net.*, (2006), http://snad.ncsl.nist.gov/nistnet/. 57

[49] Object Maganement Group (OMG), *Corba specification, v3.0.3*, (2004), http://www.omg.org/cgi-bin/apps/doc?formal/04-03-12.pdf. 6

[50] ———, *Unified modeling language (uml), version 2.0*, Specification (2006), http://www.omg.org/technology/documents/formal/uml.htm. xv, xvi, 20, 66

[51] Opensymphony, *Oscache*, (2006), http://www.opensymphony.com/oscache/. 2, 6, 39

[52] J.R.C. Patterson, *Accurate static branch prediction by value range propagation*, PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation (New York, NY, USA), ACM Press, 1995, pp. 67–78. 99

[53] S. Rubin, R. Bodik, and T. M. Chilimbi, *An efficient profile-analysis framework for data-layout optimizations*, POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (New York, NY, USA), ACM Press, 2002, pp. 140–153. 65

[54] L. Ryzhik and A. Burtsev, *Architectural design of e1 distributed operating system*, (2001), http://www.e1os.org. 43

[55] Y. Saito and M. Shapiro, *Optimistic replication*, ACM Comput. Surv **37** (2005), no. 1, 42–81. 19

[56] Richard E. Schantz and Douglas C. Schmidt, *Research advances in middleware for distributed systems: State of the art*, Proceedings of the IFIP 17th World Computer Congress - TC6 Stream on Communication Systems: The State of the Art (Deventer, The Netherlands, The Netherlands), Kluwer, B.V., 2002, pp. 1–36. 6

[57] ShiftOne, *Jocache*, (2006), http://jocache.sourceforge.net. 2, 6, 39

[58] A.J. Smith, *Cache memories*, ACM Comput. Surv. **14** (1982), no. 3, 473–530. 65

[59] Sun Microsystems Inc., *Java network launching protocol (jnlp) and api specification version 1.5.*, (2001), http://java.sun.com/products/javawebstart. 46

[60] ———, *Java web start guide.*, (2006), http://java.sun.com/j2se/1.5.0/docs/guide/javaws/developersguide/contents.html. 46

[61] Sun Microsystems,Inc, *Big heaps and intimate shared memory (ism)*, (2006), http://java.sun.com/docs/hotspot/ism.html. 38

[62] A.S. Tanenbaum and M.V. Steen, *Distributed systems: Principles and paradigms*, Prentice Hall, USA, 2002. 6, 19, 42

[63] R. Vitenberg, *Internet-wide caching of distributed objects*, Ph.D. thesis, Technion-Israel, Department of Computer Science, 2002. 8, 47, 64, 80

[64] M.N. Wegman and F.K. Zadeck, *Constant propagation with conditional branches*, ACM Trans. Program. Lang. Syst. **13** (1991), no. 2, 181–210. 100

[65] G. Yilmaz and N. Erdogan, *Dcobe: Distributed composite object-based environment*, The Computer Journal **48** (2005), no. 3, 273–291. 6, 42

[66] N. Yvan, *Jmail (source code) version 0.8.6*, (2005), http://javamailclient.sourceforge.net/html_en/index.html. 77