Effective Distribution of Object-Oriented Applications

In the last few years, the move towards distributed systems has been rapid. Distributed processing models have matured from early master/slave models to new patterns such as client/server computing and intranet computing. There is also a discernible shift in the system development paradigm from traditional, structured approaches to the OO paradigm. Several organizations have begun to move towards distributed systems and OO development in an effort to exploit the benefits of distribution, coupled with the promise of efficient development. The information technology industry has committed significant resources to addressing facilitation and standardization of distributed object computing. Virtually every important computer system and software vendor worldwide is a member of the Object Management Group (OMG) and has agreed to adopt OMG’s standard for distributed object computing: Common Object Request Broker Architecture (CORBA). Successful adoption of distributed object computing, however, will require careful attention to a number of issues, some of which include deployment and management of distributed OO systems. Finding effective distribution strategies is another area critical for adoption of distributed object computing. Though the emerging standards and other commercial tools have facilitated object distribution (see sidebar), they have not provided guidelines for effective object distribution.

Effective distribution of system components has been a concern of system designers for many years. Essentially, it involves (a) partitioning the system into appropriate distributable units, and (b) allocating these units across the existing networks of computers in a manner that would maximize some measure(s) of performance and/or minimize cost. A number of researchers have investigated the problem [3] in the context of traditional computing paradigms such as file-based or data-based system. Effective object distribution is the next step in this progression.

The research thrusts in distributed object computing has, so far, not addressed effective distribution strategies. As a recent quote from Karlapalem [4] suggests, “… no work is done in fragmentation and allocation algorithms—the major reason for this is that it is a very complicated problem.” Without appropriate distribution strategies, designers of OO systems have no guidelines for deploying their applications. With the tools and without a strategy, the situation is comparable to what a developer may face if presented with a CASE tool, with no practical guidelines on how to model an application.

This article proposes a comprehensive scheme for effective distribution of OO applications over existing networks of computers. Typically, the network of computers (for an organization) is spread across geographically dispersed sites, each containing diverse processors. The sites are linked via a communication
network, as are the processors within each site. The negligible intrasite communication costs and significant intersite communication costs define the boundaries for each site. This requires a scheme that addresses effective distribution across as well as within each site. Our proposals deal with both. Devising such a scheme to optimize meaningful design criteria is a complex problem that requires a pragmatic and opportunistic approach and demands substantial changes to the way we think about object technology. The proposed scheme exploits relevant features of OO systems and extends current data distribution research to (a) identify and formalize appropriate design criteria and (b) develop techniques for optimization of these criteria. We outline the issues faced, the pragmatic decisions required, and the solution mechanisms developed to realize the distribution scheme. We do not discuss technical details of algorithms. These are available to the readers as technical reports on the Web (see [7, 8]).

Considering OO Constructs during Distribution

It is difficult to identify distributable units of appropriate granularity from an OO system due to such constructs as encapsulation and inheritance. The different problem scopes—distributing across geographically dispersed sites and within each site—also influence the choice of granularity. Several possibilities can be identified for distributable units: classes, class fragments, method implementations, attribute

Facilitating Object Distribution

The OO paradigm adapts naturally to distributed processing. For instance, consider two objects, Customer and Order, in an order-processing application. When one object sends a message to the other, it does not need to know where that object resides or how the message will be routed. That responsibility can be delegated to an underlying implementation such as CORBA (see www.omg.org/corba2/cover.htm). The object may not be on the same processor or even at the same site—what matters is that the message is routed to the required object, regardless of its location.

The mechanisms necessary to transparently handle messaging between remote objects are substantial. However, once agreed upon, they provide the basis for design of distributed OO applications. In fact, the basic syntactic and descriptive notions of how an object will send a message to another, possibly remote, object have already been largely settled. These often take the form of a service such as the Object Request Broker (ORB). This allows developers to create a logical design of an OO application before deploying it, as desired, without requiring extensive changes. Commercial distributed object environments that facilitate deployment of OO applications in this manner are already available (www.forte.com; www.iona.com; www.dynasty.com).
clusters. A pragmatic approach to deciding which units are appropriate requires consideration of not only OO constructs, but also the different problem scopes.

Chin and Chanson [2] suggest three broad anchor points for considering the granularity continuum: large-grain, fine-grain, and medium-grain. The first indicates units (subsystems, applications) that are too coarse to exploit for effective distribution. At the other extreme, the fine-grain model suggests elements (attribute-value, object-id, line of code) that can be too numerous to consider for optimization. The medium-grain model provides a plausible compromise. It suggests meaningful application components (such as classes, class fragments, methods) as distributable units [5], that provide opportunities for effective distribution without requiring consideration of an overwhelming number of units. However, to realize this level of granularity, it may be necessary to violate strict encapsulation and stretch inheritance relations across physical spaces.

For example, physical separation of the method implementation, object instances and attribute values would undermine encapsulation. This can be especially damaging if the separation is across different physical sites, given the communication cost, longer response times, and lower levels of reliability. However, if the separation is restricted to different processors within a site in a manner that allows the best use of processor capabilities, the benefits of distribution may outweigh the problems associated with remote method invocation. Inheritance is the other key concern. Treating each class in a hierarchy separately for distribution can lead to stretching of the inheritance construct across processors or sites. Implementing this will not pose a problem since the inheritance relation can be easily maintained by giving each class knowledge of the locations of its ancestors, whether at the same site or otherwise. However, the expensive and sometimes recursive run-time probes resulting from such separation may lead to severe and unanticipated performance degradation. This has led Wegner to state that “… inheritance is incompatible with distribution …” [11]. It may be argued that the class/superclass separation can be exploited for distribution where the communication overhead is relatively insignificant—that is, inheritance coupling between classes may be preserved while distributing across sites, and exploited for distribution within each site. Table 1 summarizes our arguments.

In addition to changes in granularity, distribution criteria at the two levels can also be very different. For intersite distribution, the most appropriate concern is breaking the system into appropriate components and placing them at the required site—ensuring that the data, methods and superclasses required for processing are locally available. For intrasite distribution, the focus shifts to improving several performance criteria. This improvement can be achieved by selecting the appropriate processor type to locate each significant distributable unit. A model that addresses all these concerns in a single step would be an extremely difficult proposition to develop and use. To keep the problem formulation tractable, we propose a multilevel scheme, based on Schoeffler’s [10] strategy of decomposition by influence: “a … problem is partitioned or structured in such a way that it can be solved sequentially in levels or strata with the result or output of one stratum (a higher-numbered one) serving as partial input to another lower-numbered stratum.”

**Effectively Distributing OO Applications**

Based on the discussion in the previous section, the distribution scheme [6] can be depicted to contain two levels or strata (see Figure 1). Logical specification of the system would be the appropriate starting point for the distribution scheme. A large number of modeling notations are available to assemble the logical specification (see, for instance UML by the Rational Software Corp. [9]). This specification generally includes the class hierarchy, including associations and aggregations and specifies attributes and methods. A model of object interactions, and a description of the application functionality also forms another major component. To address distribution, we also require a description of the existing network of computers along with the system specification. Figure 1 shows an overview of the proposals that consists of two phases.

<table>
<thead>
<tr>
<th>Construct</th>
<th>Distributing across Sites</th>
<th>Distributing within a Site</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inheritance</td>
<td>Preserve inheritance coupling</td>
<td>Exploit separation of classes and subclasses</td>
</tr>
<tr>
<td>Encapsulation</td>
<td>Preserve encapsulation coupling</td>
<td>Exploit separation of methods and instances</td>
</tr>
</tbody>
</table>
**Phase 1: Intersite Distribution.** The primary focus in this phase is ensuring that the data, methods, and superclasses required for processing are available locally. Accordingly, this phase performs two tasks: (1) derivation of appropriate distributable units that reflect usage patterns, and (2) allocation of these units to sites to minimize intersite communications.

To derive appropriate distributable units we begin with class specifications available in the logical design. We mention earlier that superclasses and subclasses are best retained together for intersite distribution. To ensure this, we roll up attributes and methods of subclasses to their farthest ancestors. For example, consider a class *Customer*, with subclasses *Corporate* and *Government*. We roll up attributes and methods of the subclasses to the superclass *Customer*. This creates a package containing all relevant attributes and methods necessary for every object instance of these classes. To facilitate discussion, we have coined the term Combined Class to represent the result of this rolling up. It should be emphasized that this rollup is performed only to aid the distribution scheme, and does not involve actual promotion of attributes and methods. It ensures that inheritance is not stretched across sites. It does not, however, solve the problem of granularity. A combined class may contain a large number of instances, only a fraction of which may be required at each of the different sites. For example, salespersons in New York may need to access information only on East Coast customers. Other salespersons may only need information on government customers. Fragmenting the combined classes to satisfy specific criteria, therefore, represents a desirable strategy. Customers with *Location = 'New York'*, for example, may define a fragment of the combined class *Customer*. As another example, orders of a value of $50,000 or more, placed by customers in Florida may define a fragment of the combined class *Order*. These class fragments can be created to contain sets of object instances.

Creating such class fragments requires identification of appropriate predicates. To obtain this information we turn to usage patterns, represented as scenarios. A scenario, in essence, represents a condensed view of the message trace required to perform a typical transaction. Generally, it specifies the classes involved, an initiating site and the estimated frequency. For example, some of the functionality of an order processing application may be captured by the scenario *Generate Invoice*, as illustrated in Figure 2.

This scenario captures interactions required to generate invoices that originate in Los Angeles. The *Invoice* class serves as the initiating class for this scenario. Customer details appearing on the invoice are obtained from the *Customer* class. Information about items on the invoice is gathered from the *Shipment* and *Product* classes. Predicates for the initiating class *Invoice* are specified (in this case *Invoice_date = System_date* and *Location = 'CA'*) and propagated to other classes in the scenario. This is repeated for all scenarios of interest. Predicates (initial and propagated) are compiled for each affected attribute, for each class. Each such attribute, then, serves as the basis for generating class fragments.

Creating appropriate class fragments, however, requires a judicious selection of attributes. When multiple attributes are selected for fragmentation, the result is a superimposition of predicates. For example, selecting the *Location* and *CustType* attributes to fragment the class *Customer* will result in fragments some of which may include: *

\[\text{Location} = 'CA' \text{ and } \text{CustType} = 'Corp', \]

\[\text{Location} = 'NY' \text{ and } \text{CustType} = 'Corp', \]

\[\text{Location} = 'CA' \text{ and } \text{CustType} = 'Govt', \]

**Figure 1.** Distribution scheme overview

**Figure 2.** Capturing usage patterns through scenarios
[Location = 'NY' and CustType = 'Govt'], and so on. The number of fragments is a product of predicates available for each selected attribute. Selecting each attribute improves the potential locality of references. On the other hand, selecting too many attributes may further break required fragments and/or create too many fragments. An algorithm has been devised to generate and evaluate alternative fragments—based on different combinations of attributes—to select the combination that provides the best locality potential. Figure 3 summarizes the process described thus far. Each fragment also retains the methods. Thus, the process ensures that encapsulation is not violated and inheritance is not stretched during intersite distribution. These class fragments thus represent distributable units that need to be allocated to sites.

To allocate these class fragments to sites, we develop a mathematical programming model that minimizes intersite communications. The basic aim of the allocation model is to reduce the communication between sites. The model captures the message-intensive nature of OO applications. Inputs to the model consist of inter-fragment and fragment-site traffic volumes compiled from usage patterns. Traditionally, allocation models have assumed availability of traffic volumes, requiring a ‘guessimate’ from the designers. Instead of relying on such guesswork, we have devised a rigorous procedure that exploits the notion of reference joins for compiling these estimates. With the compiled traffic estimates as inputs, the allocation model minimizes the aggregate traffic between sites generated by fragments allocated to these sites, while ensuring availability of processor types, special processing capabilities, and storage capacity. It assumes static, as opposed to adaptive routing [1]. Many different static routing schemes are possible, especially for updates, if the class fragments are replicated. We formulate two alternatives—nearest neighbor and global update. The first captures the scheme in which an update is made by the closest available copy; the second captures the scheme in which a single copy is responsible updating all relevant copies. As the designer chooses one of the schemes, constraints and objective functions are changed to reflect the choice. The model is formulated as an integer programming problem, which can be addressed by many solvers that employ the branch-and-bound algorithm in commercial tools (see www.gams.com). Further details of formulations, along with other algorithms discussed for Phase 1, are available in [7].

**Phase 2: Intrasite Distribution.** The primary focus in this phase is optimization of several performance criteria achieved by assigning appropriate distributable units to different processor types at each site. The tasks in this phase include conversion of class fragments to appropriate distributable units for intrasite distribution, and assignment of these units over the diverse processors within each site.

The class fragments allocated to each site represent the starting point for Phase 2. Recall that these fragments are derived from combined classes—that is, they contain object instances and methods from subclasses and superclasses. These fragments do not present meaningful units of distribution for intrasite distribution, and require translation to create appropriate units for intrasite distribution. We argued earlier that class/subclass separation and violation of encapsulation should be exploited for distribution within sites, since the benefits of distribution across diverse processors outweigh the problems. Accordingly, the combined classes are broken into constituent classes, by rolling down attributes and methods from each combined class to the subclasses. The rolldown also carries with it fragmenta-

---

**Figure 3. Fragmentation procedure**

- **Start**
- **Create Combined Classes**
  - Rollup to farthest ancestor
- **Obtain Predicated for Initiating Class in each Scenario**
- **Propagate them to Other Classes in the Scenario**
  - *Any More Scenarios?*
  - Y
- **Compile List of Predicates for each Affected Class**
  - *Any More Scenarios?*
  - Y
- **Generate and Evaluate Alternative Fragments**
tion decisions to the subclasses. This converts each class fragment to one or more class fragments. Each resulting class fragment now contains a collection of object instances as well as methods from the class template. Since methods can require different processing capabilities, effective distribution requires that each method be assigned to the most appropriate processor type, that is, they be considered as separate units of distribution. However, each method need not be considered separately for distribution.

The data-related methods, typically implementing actions such as access/update of attribute values or creation/deletion of object instances, are generally coupled with the underlying database management system, and can be logically retained with the instances. User-interface methods provide the GUI capabilities and require the dedicated processing power inexpensively provided at the desktop. They are naturally assigned to platforms supporting user interfaces, and can be replicated with little penalty across these processors. Processing-related methods performing such tasks as transaction management, report generation, sorting, or other numerically intensive computation do require appropriate assignment. These may also include functionality captured by some designers as control objects [9]. The class fragments allocated to each site are thus converted to appropriate distributable units for intrasite assignment: object instance sets and processing-related methods.

The configuration of computers at each site may consist of varied topologies, connecting multiple, and possibly dissimilar, processors. For example, a typical configuration may consist of several local area networks, connected to a minicomputer, which may in turn, be interconnected via routers and a backbone network to the mainframes. It is important that the heterogeneity of processors is explicitly recognized for distribution. We accomplish this by employing the accepted notion of processor types, which keeps the model flexible to accommodate technological advances. The platforms of interest within each site thus are modeled as different processor types—to which the distributable units may be assigned. To approximate the complex configuration of processors, we employ the notion of network layers. This allows us to use the number of layers traversed by a message in the formulation of various performance measures. Since the communications costs and times within a site are relatively insignificant, the layers provide a satisfactory approximation of performance penalties involved in inter-processor messaging.

Based on this model of intrasite network of computers, we proceed to identify the distribution criteria. Choice of these criteria (see Table 2) has been shaped by our understanding of the performance issues of distributed OO systems and has been partially influenced by pragmatic modeling decisions made for this phase.

Match represents the opportunity to realize the key objective of cooperative processing. It measures how closely the processing requirements correspond to the capabilities of the processor type. It comprises several dimensions such as processor speed, primary and secondary storage, support for concurrent users, security and reliability, among others. Some of these may be hard constraints (for example, primary storage requirements), while others represent user-specified targets (such as security levels). Concurrency represents the potential of classes to participate concurrently in scenarios. For instance, in the Generate Invoice scenario (see Figure 2), the Shipment and Product classes may be needed, though not in any specific sequence. There exists the potential to access them concurrently by assigning them to different processors. An estimate of this potential can be compiled by examining all such scenarios. The third criterion, Flow, has two components. The first represents messages and responses between classes that occur as a result of their interactions in various scenarios. The second represents the interaction between instance sets assigned to different processors, estimated by examining the extent of attribute usage for each method. The last criterion, Replication, is determined by two factors: the storage cost of the object unit and the cost of updating replicated copies. This, in turn, is dependent on the size of the replicated object units, the number of copies, and the frequency of updates.

These criteria are clearly conflicting. For instance, higher Concurrency may require some redundancy, which may increase the Replication cost. Tradeoffs are also evident between Match and Flow, between

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Description</th>
<th>Desired</th>
<th>Formulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Match</td>
<td>Matching processor capabilities to requirements</td>
<td>Maximize</td>
<td>Non-Linear</td>
</tr>
<tr>
<td>Concurrency</td>
<td>Opportunities for concurrent processing</td>
<td>Maximize</td>
<td>Procedural</td>
</tr>
<tr>
<td>Flow</td>
<td>Inter-processor messaging traffic</td>
<td>Minimize</td>
<td>Procedural</td>
</tr>
<tr>
<td>Replication</td>
<td>Cost of replicating object instances</td>
<td>Minimize</td>
<td>Linear</td>
</tr>
</tbody>
</table>
complicate matters further, the criteria (other than Replication) require complex formulations that are not amenable to algebraic manipulation. Trade-offs among the criteria are also difficult to articulate since they can vary at different value ranges. To adequately address these issues, we develop a satisficing procedure by extending conventional MCDM principles (see Figure 4).

The procedure begins by generating and evaluating a large random sample of alternatives to characterize the search space for each criterion. Since the different criteria have different units of measurement and can produce widely varying value ranges, the evaluations are normalized and inverted where necessary. This allows assessment of each alternative on the basis of criteria scores that vary from 1 to 100—the higher the score, the better the alternative for that criterion. The scores obtained from the large sample are used to construct a search space for each criterion, in the form of a graph (see Figure 5). Presenting the information in this manner allows the system designer to grasp the totality of the problem and provides the necessary context for making judgments and trade-offs. Also, as the sample is generated, a set of promising solutions is retained for further analysis. These represent solutions that have a score higher than others on at least one of the criteria (non-dominated solutions [12]).

The iterative search procedure that follows is jump-started by these nondominated solutions. They provide the initial seed points that identify promising regions of the search space. At each iteration, the search involves complete exploration of the neighborhood around a solution—within tolerances provided by the designer. For example, given the scores of 55, 73, 42 and 83 for the four criteria, the designer may decide to search the neighborhood by specifying the tolerances as +5, -2, +10 and -3. By adjusting the tolerance levels, the designer can travel towards the desired pareto-optimal solution. During each such probe, nondominated alternatives that fall within tolerances are retained. At every iteration, the designer can use a new seed point or any of the retained alternatives for further probes. Using a new nondominated solution to start each probe also ensures that efforts are not wasted in analyzing clearly inferior solutions. If a region appears unpromising, the designer can discard it and initiate another thereby reducing the possibility of becoming locked within a local region.

The distribution procedure aids in this search by giving feedback. Summary information allows the designer to judge the quality of the current alternative relative to other alternatives generated. An estimate of the likelihood of further improvement is also computed using probabilities from the search spaces which allows the designer to judge the quality of the current alternative relative to other, unexplored alternatives. As an example, consider an alternative with scores 61, 71, 69 and 81. With the set of search spaces generated earlier, the probability of generating another alternative that does better on all criteria may be estimated at, say, 2%. Since all alternatives cannot be explored, this feedback provides the designer some confidence in the solution obtained. The decision to stop the search rests with the designer, as does the final selection of the pareto-optimal solution [12]. Detailed technical formulations of the performance criteria and the MCDM-theoretical and statistical bases of the decision support procedure were omitted here. Complete details of both are available in [8].

Object Distribution Environment

A prototype ODE has been implemented to verify the proposed object distribution scheme. The algorithms for Phase 1—class fragmentation and intersite traffic compilation—have been custom-coded in C on an Intel PC; the integer programming models for intersite allocation have been designed using the General Algebraic Modeling System (GAMS) (see www.gams.com) on a SUN 4/SPARC. The distribution criteria for intrasite distribution have been coded in C as well, and the decision support procedure has been implemented in ObjectPAL(tm), a 4GL on an Intel PC. Figure 5
shows snapshots of sample screens from ODE. Here, the screens show a session in progress for intrasite distribution.

The distribution scheme was verified by using it to distribute a marketing information system of a midwestern utility company. The technical sources cited earlier [8, 9] provide a detailed description of the application and the results obtained. Briefly, the techniques developed for Phase 1 were executed successfully for derivation of class fragments and allocation to sites to minimize intersite communications. For Phase 2, a pareto-optimal solution with scores of 69, 91, 74 and 76 (compared with averages of 52, 45, 32, and 46, respectively) was achieved for the four performance criteria at the test site. Implementation of the prototype and application of the scheme indicated that the proposed scheme is viable and useful.

**Conclusion**

As distributed object computing standards (for example, OMG’s CORBA) have matured, they have paved the way for object deployment tools such as Orbix, Forte, and Dynasty (see sidebar). This has made distributed object computing a reality for many organizations. Successful adoption of distributed object computing, however, will require careful attention to a number of issues, some of which include deployment and management of distributed OO systems. Effective distribution strategies is another area that is critical for adoption of distributed object computing. In the scheme described in this article, we have made a pragmatic start towards developing usable solution approaches for effective object distribution.

The scheme consists of two levels. The first specifies procedures and algorithms that can be used to distribute OO systems across geographically dispersed locations—in a manner that would maximize locality of references. The second level formulates several performance criteria and an iterative decision support procedure that can be used to derive a pareto-optimal solution for distribution of OO systems in local environments that support the client/server or cooperative processing model. The inter-site distribution mechanisms will be particularly useful for business-oriented (rather than scientific) applications, which contain significant amount of transaction data, that is, a number of object instances for some classes. A similar comment may be made, though to a lesser degree, for the intrasite distribution scheme. Application designers may use the mechanisms described at either level separately, say for distribution across sites only or within a site only—or may follow the entire scheme. Technical reports that contain the algorithms, procedures, mechanisms and models described are available to application designers [8, 9].

For the research community, this project has investigated distribution issues in the context of OO systems. In particular, it has demonstrated how traditional techniques (from data distribution research) can be enhanced and new techniques developed for effective distribution of OO applications in existing computer networks, both across and within sites. The work completed thus far has been instrumental in gaining substantial understanding of the complexities involved in the process. The work has also confirmed the inherent complexity of the object distribution problem. There are several key problem
areas where further research is warranted. First, new fragmentation schemes that consider couplings in OO systems (such as attribute-method and aggregate-component) are required. Second, formulation of additional design criteria that are meaningful for OO systems (for example, ease of software updates) is also needed. Finally, an extension of the object clustering research stream to business-oriented applications (which demand different performance measures) is required.

REFERENCES

This research was supported, in part, by a research grant from College of Business Administration of Georgia State University.

SANDEEP PURAO (spurao@gsu.edu) is an assistant professor of computer information at Georgia State University, Atlanta.

HEMANT JAIN (jain@csd.uwm.edu) is a research professor of management information at the University of Wisconsin-Milwaukee.

DEREK NAZARETH (derek@csd.uwm.edu) is an associate professor of management Information at the University of Wisconsin-Milwaukee.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

© 1998 ACM 0002-0782/98/0800 $5.00