CHAPTER VIII

FUTURE WORK AND CONCLUSIONS

The algorithms, techniques, and tools for separating distribution concerns, explored by this dissertation, present ample possibilities for future work. Each of the developed software tools can be further enhanced in terms of both its capabilities and applicability. Furthermore, some of the general insights gained from this research can be applied to domains other than distributed computing. Some of the future research directions, resulting from this work, have already been explored both by us [92] and other researchers [107]. We next present some of the ideas for future work for NRMI, GOTECH, and J-Orchestra. After discussing the future work directions, we reiterate the merits of this dissertation and present our conclusions.

8.1 NRMI Future Work

NRMI, with its call-by-copy-restore semantic that makes remote calls look like local calls for stateless servers and single thread clients, is a convenient building block for other middleware facilities that emphasize ease of use without jeopardizing performance. Specifically, we would like to take our work on NRMI in the directions of greater generality and adaptability to network outages. The first direction will extend NRMI to explore a general problem of efficiently synchronizing a subset of the client state against a subset of the server state by means of a remote procedure call. This problem is common in enterprise

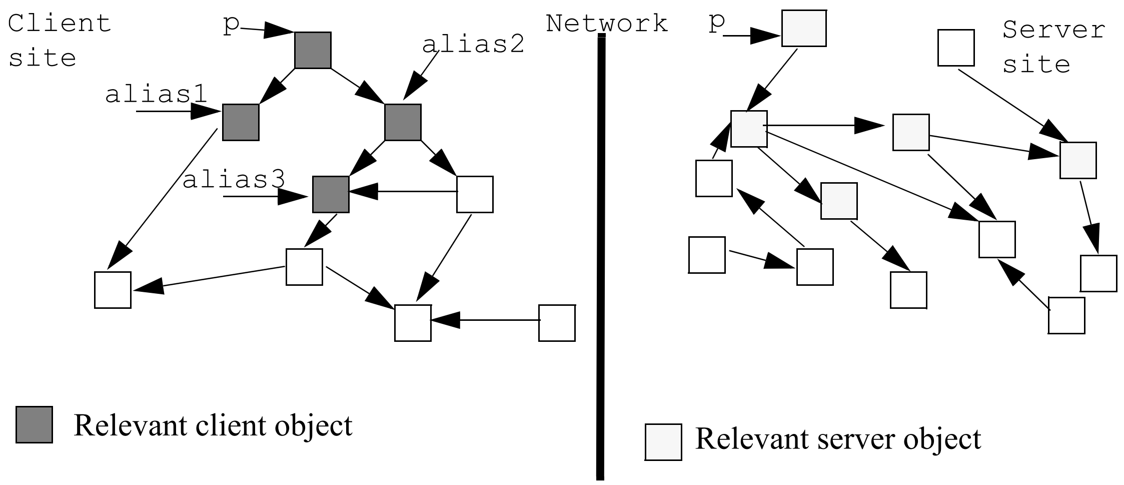

Figure 8-1: (a): A general remote call mechanism: a subset of the client heap, reachable from `p`, can be sent to the server, to be updated against a subset of the server heap.

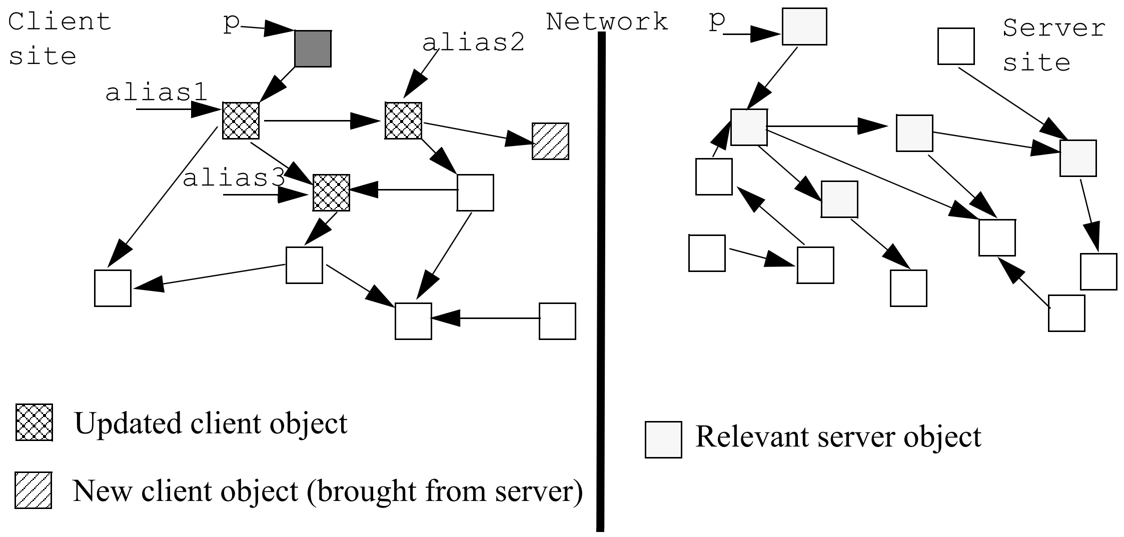

Figure 8-1: (b): A general remote call mechanism: param `p` is returned to the client and restored in place.

computing such as the domain captured by the J2EE specification [78]. Often a server environment contains a very large state of heap-allocated objects and clients need to synchronize themselves periodically against this state. Currently, no existing mainstream technology provides a convenient programming mechanism that implements this functionality. As a result, programmers resort to ad-hoc solutions that are error-prone and difficult to maintain and extend.

In abstract terms, the solution can be provided by an efficient implementation of a relaxed version of call-by-copy-restore semantics. For lack of a better term, we will call the mechanism that implements this semantic a “general remote procedure call.” Figure 8-1 (a and b) demonstrates the desired behavior. One can provide an efficient implementation of a general remote procedure by reusing NRMI with its ability to update objects in place (i.e., preserving all the aliases) and extending it with a customizable serialization mechanism. NRMI already relies on Java Serialization [79], which provides the `transient` keyword to indicate a field that is not part of an object's persistent state and should not be serialized. Nevertheless, the `transient` keyword is too crude a mechanism for providing a truly customizable serialization as would be needed by the general remote procedure call mechanism. One interesting question that is to be explored is how this customizable serialization mechanism can be best expressed by the programmer. Perhaps it can be accomplished through special purpose annotations or even via the means of a domain specific language (DSL). Another question is how easy would that be to integrate this general remote procedure call mechanism into an application server environment such as the one provided by JBoss. Finally, it would be important to determine whether a general remote procedure call implementation can be optimized enough for real-world use.

Another future direction for NRMI work would be providing an adaptable middleware mechanism that could respond to network outages. This mechanism would have the potential to enhance data availability and the overall quality of service (QoS) in unreliable networks such as dynamic mobile wireless networks. Because of their ad-hoc nature such networks are volatile and can temporarily become disconnected. Furthermore, usually outages in such networks are temporary and short in duration. If such a network outage happens during a remote call, the client computation might proceed up to the point when the data returned by the remote call is first referenced, which might be at some later point in the control flow than immediately following the remote call. This mechanism would enable continuing computation while the network is temporarily unavailable, and, in the presence of frequent but short interruptions, can result in improved throughput. Of course, this adaptable mechanism would be applicable to regular call-by-copy semantics in remote calls as well, taking into consideration only the return value of the call. However, with call-by-copy-restore, which also changes the values of the parameters of a remote call upon return, the problem becomes more comprehensive—solving it would require taking into consideration all the variables that could have been changed as a result of a remote call.

At the implementation level, realizing an adaptable middleware mechanism that could respond to network outages will require a combination of static analysis and code rewriting. Such static analysis techniques as control flow can determine conservatively the actual statements referencing by-copy-restore parameters and the return value of a remote call in the client portion a program that follows it. Then the code can be automatically rewritten to delay the blocking, occurring as a result of a temporary network outage during a remote call, up to the program statements determined through the analysis. Of course, extensive benchmarking at both the micro and macro levels would be required to determine how successful such a middleware mechanism can be in improving the throughput of applications operating in volatile network environments.

8.2 GOTECH Future Work

The GOTECH framework is one of the first research projects that has taken the approach of combining generative and aspect-oriented techniques. The GOTECH approach can be enhanced in several directions such as improving the framework and providing tools that would make it applicable in domains other than distribution. A representative of the latter direction is a recent work by my colleagues on a generator called Meta AspectJ (or MAJ for short) [107]. Their work has successfully resolved one of the major shortcomings of the GOTECH approach—its reliance on text-based templates.[1] As an evolutionary improvement of the GOTECH approach, MAJ represents the generated code as a typed data structure instead of arbitrary text, generating syntactically correct AspectJ programs. While the MAJ project has focused on providing a general-purpose generator, it would be interesting to explore how well a combination of generative and aspect-oriented techniques can help solve problems in domains other than distribution, with persistence and security being most promising.

Another future direction would be providing more mature support for the conversion of plain objects to EJBs with different tools. For instance, the JBoss AOP framework performs bytecode engineering at class load time to retrofit existing classes so that they become EJBs. This approach can be applied both to distribution and to persistence concerns and is of high industrial value. Since NRMI has already been implemented to work with JBoss, this bytecode engineering work can result in a replication of the GOTECH capabil-

1. Reliance on text-base templates is not a serious issue for GOTECH per se, which is a domain-specific generator with a fixed set of templates, but it definitely becomes so for any software generator that aims at generalizing the GOTECH approach.

ities at load-time. Finally, another promising direction for more mature use of GOTECH includes developing analysis tools that formalize the preconditions for the applicability of the approach and ensure they are met by a specific application.

8.3 J-Orchestra Future Work

J-Orchestra is the largest and most comprehensive software tool for separating distribution concerns explored by this dissertation both in terms of the actual distribution concerns that it successfully separates and in terms of the various case studies to which it has been applied. It is natural, therefore, that our work on J-Orchestra has led to multiple and diverse future work directions. Since it would be unrealistic to describe all of these future work directions in detail here, we outline some of the major ones next. These directions fall into two major categories: expanding the boundaries of application partitioning and applying the insights gained from the J-Orchestra project to domains other than distribution.

While J-Orchestra has demonstrated that automatic application partitioning is a viable technology for introducing distributed capabilities to a specific class of centralized applications, future work can address various limitations and shortcoming of the J-Orchestra approach. One inherent limitation of J-Orchestra has to do with the automatic nature of its approach. That is, the J-Orchestra user works at the class or group-of-classes level of abstraction. Thus, our approach is quite automatic and involves no programming, just resource-location assignment—for example, that graphics code should run on this machine, or the main engine should run on that machine. In contrast, a semiautomatic approach could let the user annotate detailed parts of the code and data, to indicate, for example, what data should be replicated, how the copies should remain consistent, and how leases should be used for fault tolerance. Thus, a semiautomatic approach could resolve many of the issues associated with automatic partitioning.

One of such issues is that, in its present state, automatic partitioning does not offer any assistance in supporting highly dynamic interactions between communicating entities, which are common in ubicomp applications [97]. For example, ubicomp applications often allow for resources and services to come and go dynamically as users and devices enter and leave the environment. Because automatic partitioning does not change the original centralized application's logic or structure, flexibility and configurability must be designed into the original application before it is partitioned. In contrast, a semiautomatic approach could potentially support dynamic interactions through automatic modification of an unsuspecting application.

In general, a partitioning system tries to automate many hard distribution tasks. Any automation effort, however, hinders complete control for users with advanced requirements. Such requirements might include replication for fault tolerance; high performance through load balancing, caching, or asynchronous communication; security; and persistence. In an automatically partitioned application, it is not easy to use replication for redundancy and switch to a different server once a failure is detected. The conventional wisdom in the distributed-systems community is that mechanisms for handling distributed failure are extremely application-specific and can not be automated completely.

Again, the appropriate solution might be to follow a semiautomated approach, providing tool support for replication, load balancing, security, and so forth. In this way, the programmer would be relieved of the low-level complexity but would still be responsible for annotating parts of the code in detail and for the distribution's conceptual consistency.

In fact, Section 4.5.4 has described how J-Orchestra supports a semiautomated approach that enables the user to specify complex schemes for object mobility (e.g., “move this object whenever it is reachable from an argument of a remote method”). Nevertheless, because this is not a GUI-accessible feature, the user must write Java code that follows J-Orchestra framework conventions to enable such object mobility.

At the implementation level, a semiautomatic approach could, for example, let the user annotate the application code to express desired policies for data consistency in the context of possible failures. These annotations would form a domain-specific language for specifying properties of dynamic distribution. For instance, one could annotate a certain data field to indicate that many instances of it might exist. Another annotation could specify the leases that each client holds and the data that depend on each lease. The low-level code would then be generated from the annotations instead of having to be handwritten. Overall, the approach would be very similar to the one currently followed by the GOTECH framework, but it would also involve the J-Orchestra analysis and bytecode transformation engines, making it more powerful.

J-Orchestra currently uses a type-based analysis heuristic that determines which references can leak to which code. This heuristic is too conservative and its precision and sophistication can be improved. Specifically, one promising direction would be to expand it with various static analysis techniques such as control-flow and data-flow that would help determine with a greater degree of precision which references can leak to which native code. Of course, as we demonstrated in Chapter VI, any solution to this problem would be an approximation, and one has to make reasonable assumptions to account for both the inherent limitation of the existing static analyses techniques and the unpredictability of native code behavior. Nevertheless, a more sophisticated analysis engine would enable more powerful rewrites.

One of such rewrites could support object-based partitioning, which would be orders of magnitude more fine-grained than the current class or group-of-classes abstraction level at which J-Orchestra operates. Of course a full object-based partitioning approach would not scale to realistic applications, but, when applied to only a limited subset of classes, it would be an extremely valuable addition to the existing J-Orchestra tool set. J-Orchestra already supports a limited version of object-based partitioning based on the objects' creation sites. Nevertheless, in this case, it is entirely up to the programer to ensure that such object-based partitioning makes sense. An object-based analysis could provide the programmer with information about how particular objects are used in the program, enabling more sophisticated partitioning scenarios.

Another tool that could empower the programer in making more informed partitioning decisions is the J-Orchestra profiler. In its current stage, the J-Orchestra profiler provides a very crude kind of information and as such offers several directions for future work. An important issue with profiling concerns the use of off-line vs. on-line profiling. Several systems with goals similar to ours (e.g., Coign [33] and AIDE [56]) use on-line profiling in order to dynamically discover properties of the application and possibly alter partitioning decisions on-the-fly. So far, we have not explored an on-line approach in J-Orchestra, because of its overheads for regular application execution. Since J-Orchestra has no control over the JVM, these overheads can be expected to be higher than in other systems that explicitly control the runtime environment. Without low-level control, it is hard to keep such overhead to a minimum. Sampling techniques can alleviate the overhead (at the expense of some accuracy) but not eliminate it: some sampling logic has to be executed in each method call, for instance. Another issue has to do with fine-tuning the technique for analyzing the profiling results. This technique, given some initial locations and the profiling results, should determine a good placement for all classes. The technique that J-Orchestra currently follows is a clustering heuristic that implements a greedy strategy. It would be interesting to experiment with replacing this clustering heuristic with other algorithms that could provide a reasonable approximation, particularly for the situations when the number of partitions is greater than two.

In its current implementation, J-Orchestra treats security as an orthogonal concern. On the one hand, in designing the J-Orchestra rewrite engine, which transforms a centralized program into its distributed counterpart, we have made every effort not to introduce security vulnerabilities if at all possible. At the same time, we did not have an opportunity to have our rewrites follow a well-defined security policy. Producing a secure distributed system as the partitioning's end product has not yet been one of the primary objectives of J-Orchestra. Nevertheless, partitioning presents many interesting security challenges. Some prior work had focused on secure program partitioning [104], which is different from the problem of applying a security policy to resource-based partitioning. By splitting up the functionality of a centralized program to run on multiple network sites, talking to each other over the network, some information that would have never left the confines of a single address space, suddenly can get transferred over the network. Producing a coherent security policy and incorporating it into each and every step in the partitioning process could be an interesting research direction.

Aside from distribution, some of the insights, gained from our work on J-Orchestra, can be generalized to other domains. In abstract terms, the J-Orchestra approach can be described as adding capabilities to existing programs through bytecode modifications. In the case of J-Orchestra, the added capabilities are distribution. In the past, bytecode manipulations have been used to add other capabilities to existing programs, including persistence, profiling, logging, and so forth. Nevertheless, our work on J-Orchestra has achieved results that distinguish it from other work in its handling of the bytecode/native code interactions in the runtime system. Therefore, it would be beneficial to generalize our techniques from the domain of distribution to other domains, particularly the ones that have already employed bytecode transformations in the past. The indirection machinery of J-Orchestra can be generalized in a completely domain-independent way, resulting in a tool that would allow adding capabilities to existing programs by modifying the bytecode not only of application classes but also of system classes, whenever possible. One interesting application of this tool would be extending AspectJ with capabilities to apply aspects to systems classes.

In general, applying bytecode transformations can yield benefits in a variety of domains and software development scenarios. We have attempted to generalize the technique by exploring the idea of binary refactoring [92], which applies refactoring transformations (e.g., split class, glue classes, inline method, remove design pattern indirection) to a software application without affecting its source code. Binary refactoring is only the tip of the iceberg, but it demonstrates an important principle that a good program transformation approach should follow: program transformation should not sacrifice software maintainability in order to achieve performance or temporary convenience. It would be interesting to see how program generation and transformation can be applied to large-scale program modifications.

8.4 Merits of the Dissertation

This dissertation has explored algorithms, techniques, and tools for separating distribution concerns. We discussed the motivation, design, and implementation of three software tools: NRMI, GOTECH, and J-Orchestra. We also identified the applicability issues of these tools and presented validation through case studies. We next reiterate some of the conceptual contributions of this dissertation.

1. *A general algorithm for call-by-copy-restore semantics in remote procedure calls for linked data structures.* The NRMI middleware mechanism provides a fully-general implementation of call-by-copy-restore semantics for arbitrary linked data structures, used as parameters in remote procedure calls.

2. *An analysis heuristic that determines which application objects get passed to which parts of native (i.e., platform-specific) code in the language runtime system for platform-independent binary code applications*. The J-Orchestra system utilizes this analysis heuristic to enable partitioning of unaware programs in the presence of unmodifiable native code in the runtime system. We also discuss how this heuristic can be fine-tuned and applied to other domains.

3. *A technique for injecting code in platform-independent binary code applications that will convert objects to the right representation so that they can be accessed correctly inside both application and native code*. The J-Orchestra system implements this technique in its rewrite for classes with native dependencies.

4. *An approach to maintaining the Java centralized concurrency and synchronization semantics over remote procedure calls efficiently.* The J-Orchestra system follows this

approach to transform centralized concurrency and synchronization Java constructs for distributed execution.

5. *An approach to enabling the execution of legacy Java code remotely from a web browser*. This approach is called appletizing, and it is fully realized as a specialization of automatic partitioning in the J-Orchestra system.

8.5 Conclusions

This dissertation has discussed research that is concerned with developing and evaluating software tools for separating distribution concerns. The goal of this research is to introduce software tools working with standard mainstream languages, systems software, and virtual machines that effectively and efficiently separate distribution concerns from application logic for object-oriented programs that use multiple distinct sets of resources. We believe that this research will contribute to the development of versatile tools and technology with practical value, innovative designs, and the potential to become mainstream in the future.

It is an exciting time to be a researcher in the field of software technology. For the first time in the history of computing, we have mainstream commercial languages such as Java and C# that are virtual machine based, platform-independent, garbage-collected, fairly type safe, conducive to good software engineering practices, and easily amenable to code transformation and generation. In addition, programs written in these languages show good and improving performance, thanks to the ever more sophisticated Just-in-Time compilation technologies. As a consequence, many interesting research developments in software technology, before applied to and tested on exclusively esoteric, research-only language environments, will be transferred to mainstream software development at ever accelerating rates. All this makes software technologies a highly-dynamic research area with the potential of influencing how we build software today and in the future, and, hopefully, the contributions of this dissertation are a concrete step in realizing this vision.