AINA2011 - IEEE 25th International Conference on Advanced Information Networking and Applications, Singapore. 22-25 April 2011.

Simple, Weakly-coupled, Invisible Middleware (SWIM)

Martin Bateman
CEPS
University of Central Lancashire
Preston, UK
mbateman@uclan.ac.uk

Saleem Bhatti
School of Computer Science
University of St Andrews
St Andrews, UK
saleem@cs.st-andrews.ac.uk

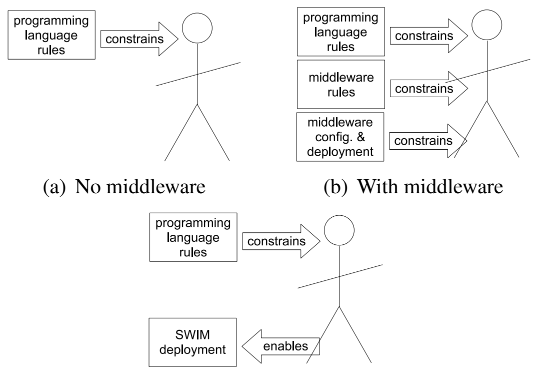
Abstract

One of the operational goals of a middleware platform is to provide a mechanism of distributing computation requests in a way that hides from the programmer the complexity of the underlying systems platform. This means that distribution mechanisms used to harness a set of computer and network resources should not expose to the programmer the detailed systems aspects which are unrelated to their application. Ideally, the programmer should be left to concentrate on the functionality of his/her application without having to be concerned with how the distribution is achieved or how the resources are used. However, this is not true today: programmers need to be aware of details of the middleware in use and are constrained by it in the design of their application, e.g. API constraints. We present a proof-of-concept demonstration of a middleware platform that imposes absolutely no constraints on the programmer apart form those used in the programming language itself. We demonstrate the efficacy of our approach with a prototype implementation in Java, running on a cluster of 20 nodes with a performance comparison with XML-RPC and Java-RMI.

1 Introduction

We focus on one key role of middleware: use of distributed services. From a programmer's perspective, the distribution of the resources and the mechanisms that enable the distribution should be invisible. However, middleware today often lacks this transparency. Indeed, when the programmer is exposed to some of the underlying middleware system behaviour s/he may be able to optimise his/her application for operation using that middleware, e.g. for performance or error reporting. Our approach aims to investigate the least constraint that we can impose on the programmer within a middleware system.

In Figure 1, we depict the proof-of-concept which we wish to demonstrate. We assume that the programmer has a design which needs to be implemented, and this subject to constraints from the programming language chosen 1(a). This is true whether the application will be engineered using middleware or not. In reality, there are many middleware technologies which aim to reduce the complexity of creating distributing applications but these systems typically enforce some requirements from the middleware platform into the application design and engineering, i.e. the programmer is constrained further 1(b). These constraints range from the types of data that you are able to transmit, as in the case of XML-RPC, or which components of the system are to be remotely hosted.


programming language rules
constrains
middleware rules
constrains
middleware config. & deployment
constrains
programming language rules
constrains
SWIM deployment
enables

(a) No middleware (b) With middleware

(c) The aim of SWIM

Figure 1. A programmer's perspective

The net result is that the application logic and the distribution logic are entangled within the code design and implementation. So, the use of middleware has some side-effects:

- The programmer requires some knowledge about the middleware platform that is chosen.
- There may be design and engineering constraints imposed on the programmer by the middleware platform.
- The application becomes dependent upon the middleware platform chosen.

There could be other side-effects, but these are the ones that are the focus for the work presented here.

Our aim in this paper is to investigate to what extent it is possible to allow the programmer to write applications that:

- Do not require the programmer to have knowledge of the middleware.
- Remain disentangled from the details of the middleware.
- Are not dependent on a particular piece of middleware to operate.

i.e. the programmer *enables* distribution of the application, as required, through our *Simple, Weakly-coupled, Invisible, Middleware (SWIM)* (Figure 1(c)).

To demonstrate our proof-of-concept middleware platform, SWIM, we first outline the scenario we have implemented (Section 2). We consider related systems against which we have tested our implementation and identify in more detail the constraints that we wish to avoid (Section 3). We describe the architecture for our demonstration of SWIM (Section 4) and present the testing and evaluation of SWIM against two popular platforms (Section 5). We discuss the limitations of the SWIM proof-of-concept, and possible solutions (Section 6). We then conclude and give a brief explanation of our plans for future work (Section 7).

2 Requirements and scope

In this paper, we take the position that the programmer should focus only on his/her application and not be concerned with any issues related to distribution[1] So, our scenario for this paper is based on the assumption that a program can be written as if it was to run on a on a single machine, but will, through SWIM, run in a distributed manner. Such transparent distribution of an application allows for minimal programmer effort whilst still gaining the advantages of distributing an application in this way, for example performance increases from parallel processing on many nodes or increased reliability.

2.1 Scope – a testbed

To provide a tractable scenario for implementation, we choose a very simple testbed. However, we will discuss in Section 6 the limitations of this testbed, and how the SWIM proof-of-concept could be extended to deal with such limitations whilst maintaining the aims listed above.

[1]Our approach in this paper is deliberately user-centric (programmer-centric). We leave for another discussion the approach from a systems engineering viewpoint.

Our testbed consists of a cluster of 20 compute nodes, all of identical capability. The nodes are all on a single network, managed within a single, trusted administrative domain. This might be typical of an in-house compute cluster at an end-site. Two examples are a dedicated computer cluster for processor intensive tasks at a commercial site, or the use of a lab/teaching facility with identical desktop machines.

The intention is to demonstrate the use of a test program written to run on a single node, and then show how SWIM allows it to run across all 20 nodes, without changing any of the code. The program is written in Java, which allows a comparative test against XML-RPC[2] and Java-RMI[3].

The SWIM proof-of-concept concentrates on how the *distribution* of the code can be achieved in a transparent manner across the cluster:

- Code is written only once as if middleware was not in use, i.e. using only the code written by the programmer and any third party libraries required for the application logic.
- All Exceptions that are defined are honoured, across the distributed platform.
- No special design or engineering is included that assumes the program will be run in a distributed manner.
- No special error-handling is required to trap errors that may result from the application being distributed or having to communicate over a network.
- No modified or special compiler tools are used: only standard tools available with the normal distribution of Java are used.

There are some platform-specific issues we have factored out in this study for the sake of focusing our analysis on the feasibility of the approach. We list them now to make clear the functions that are not considered for the SWIM proof-of-concept, but these are all issues that would be considered important for an operational middleware platform.
Security: Effectively, we factor out any security, audit/access and authentication issues:

- All the nodes are trusted by the programmer, and the programmer is 'trusted' (has the correct privileges to use) all the nodes.
- All the code that will be run is 'trusted', implicitly: no checks will be performed (e.g. checking certificates) for any of the programmer's own code, or any libraries used.

[2]http://www.xmlrpc.com/spec
[3]http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html

Fine-grained resource control: there is no constraint on the way the resources are allocated, though they are allocated dynamically, rather than as fixed quanta. We will only have a single user on this cluster for our evaluation, so no sophisticated resource control is required:

- All the programmer's resource requests will be permitted by any node.
- All the nodes will operate with the same same resource usage policy.
- The programmer has no capability to optimise from the user code the use of the resources within the cluster.

Platform-specific error-Handling: there is no specific error reporting and handling that is related to SWIM:

- Errors related to the operation of SWIM can not be reported to or handled within the application.
- The semantics and effects of errors arising as the result of distribution may be lost, especially if the errors are from faults related to the platform itself, e.g. resource exhaustion and node failures.

In [8] Spiegal argues that are some solutions to partial failures, which are able to mask the application logic from the failure. These solution are independent of the applications and can be performed by the middleware and are therefore transparent to the application.

2.2 User (programmer) requirements

From the point of view of the middleware user (programmer), there should be benefits compared to a non-distributed application, but transparency should not be lost, i.e. the use of the middleware should be invisible to the programmer. So, we concentrate in this study on benefits to the user, specified as requirements in such a way that the user is unaware of how they are implemented or how the operate.
Code usage: Applications are typically a combination of new and existing code, where the existing code could be legacy code or it could be software components which provide specific functionality. It should be possible to include these software components in the distributed system with minimal or no modification.
Dynamic resource usage: Many applications are dynamic in nature, with changing demands not only in the long term, for example over the life of the application, but also in the short term. If we take a simple example of a network authentication service, there is a daily cycle of load requirements, with a higher load required at the beginning of the day, when many people are using it to authenticate themselves. Traditional systems require provisioning for this peak demand which is then under utilised for the rest of the day when there are very few authentication requests. The system dynamically uses resources from the cluster as required. Of course, the application domain for which such distribution is suited is for tasks where the function lends itself easily to parallelisation.
Distribution without added complexity: A remote call can behave differently from a local call. Of course, in terms of the time it takes to make the call a distributed system may take milliseconds rather than nanoseconds, for example. However, remote calls introduce a new set of potential error conditions which must be handled. There is little which can be done to reduce the invocation time of a remote call (other than protocol optimisations) but handling the error conditions to some degree is possible within the middleware by using object replication, and so not exposing them to the application.
Distribution without code changes: Traditional, existing middleware systems require explicit distribution points to be defined, for example using IDLs in CORBA or interfaces in RMI. Once defined, these boundary points cannot be changed without rewriting parts of the application. This can require that the object to be distributed must subclass from a specific object, which in the case of an existing application may require extensive re-engineering. Then, anywhere in the code that the new remote object is used must be altered in order to deal with any error conditions relating to the network. This can require extensive modification of an existing application since anywhere that the new remote object is used must be modified.
Fault tolerance: Within a distributed application, it can be necessary to move objects from one host to another. This could be for purposes of robustness in the face of failures or for maintenance. Performance gains could be gained from moving objects geographically closer to the client, or to a more resource capable node, or in order to reduce the round trip time and, therefore, the time it takes to make a remote call. However, we do not consider such optimisations in this study. Instead we aim only to make transparent any maintenance or node failure events: any hosted objects need to be relocated if the node on which they are running goes out of service.

3 Related Systems

Traditional middlewares such as XML-RPC, RMI or CORBA require that the distribution logic and the application logic are intertwined. Typically the middleware enforces some design requirements, be it restrictions on the types which can be used, the use of interfaces, or the requirement to subclass from specific superclasses. Each enforces a slightly different set of restrictions and requires a slightly different steps in the development process. As such these technologies are difficult to use and error prone, whilst adding little to the application logic but enforcing design decisions. Decisions concerning the distribution points of the application must be made at design time and are impossible to change at deployment.

Lee & Morris [7] proposed a system of parallel computing using the data-flow model [2]. Their system requires a change to the programming style imposing a set of restrictions based on the toolkit. Thiruvathukal et al [9] augment the Linda [3] and MPI [4] styles of middleware, but again requires a specific programming style to be adopted.

Dynamic adaption has been proposed by many (for example in [1]) as the long lived applications may have to evolve over time in order for applications to change to meet the demands of the changing environment. Zhu et al [11] proposed using lightweight threads for load balancing. Their solution allows for thread migration at chosen migration points within a method, requiring both a modified JVM and the modification of the application to be ported to their distributed JVM (dJVM).

Dearle et al [5] share the same vision of providing a transparent distributed platform although they require that the programmer define distribution points when the application components are remotely hosted meaning the programmer still has to be involved, albeit lightly, with the distribution process.

Luo et al [6] use a distributed JVM but unlike other systems [1, 2, 7, 9, 11] there system does not require a modification to the JVM or a change in programming style and is therefore able to operate on the standard Sun JDK with minimal programmer intervention.

Many of these system have similar goals - to decrease the complexity for the programmer of the system but adopt different approaches for achieving this goal.

4 SWIM

The SWIM system has three systems, a registry, a number of SWIM nodes which form the SWIM compute pool and the client application. When starting up each SWIM node registers itself with the SWIM registry.

4.1 Architecture

A SWIM node provides the object hosting environment with the SWIM pool. It consists of a custom classloader which is able to load classes from across the network using SWIM as the transport medium. The URLClass loader was avoided since it would require a web server to be running on each SWIM node as well as the SWIM server. The connection handler is responsible for taking incoming request from the network, looking up the hosted object and making the method call.

The registry within SWIM provides a way for clients to find the nodes or to find objects which have been specifically hosted by the application programmer. The only information stored at the registry is the name of the hosted object, which classes and interfaces it provides and its address.

All SWIM nodes are referenced using a socket address, a non transport mechanism addressing scheme. This, coupled with the fact that the connection handler uses streams to communicate means that a SWIM node can be addressable from a wide range of transport protocol. The default is a standard network socket, but we also had support for peer-2-peer sockets [10] and SSL sockets. Each of these three is designed to provide a specific functionality to the application. The use of a peer-2-peer infrastructure allows the SWIM pool to be distributed across a heterogeneous network but at the expense of execution speed since all messages are routed around the peer-2-peer overlay.

The SWIM client takes one of several forms. In its simplest incarnation it is a series of APIs which allow the programmer to host objects within the SWIM pool. These are used list a traditional middleware platform. The SWIM API includes methods to host an object within a SWIM node, as well as method to clone and migrate objects from one SWIM node to another. When an object is migrated a tombstone is left which redirects any requests to the object to the new SWIM node.

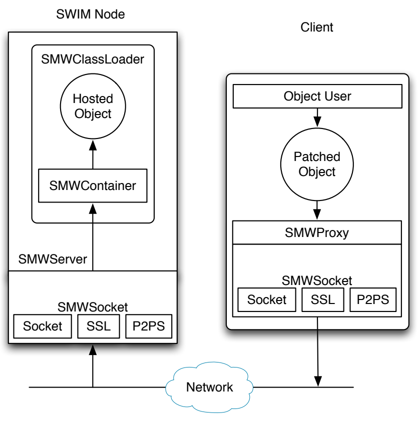


Figure 2. The SWIM Architecture

The second form of interaction is via the transparent distribution classloader, this allows either complete delegation of hosting decisions to be taken by SWIM or guided decisions where the programmer supplies a configuration file stating which objects should be hosts. When an application starts each object is loaded by a custom class loader. If the object is to be remoted, which is determined by a deny and allow list, then the objects byte code is cached to later use. Each method within the object is replaced with a method which makes a call to the remote object. The constructors are replaced with code which creates an instance of the local object in the remote hosting environment. If an object of the correct type does not exist within the remote hosting environment then the local cached class file is uploaded to the remote hosting environment and is used. Within the hosting environment each hosted object is loaded by its own classloaded, there is one classloader per hosted object. This means that even if an object with the same name has previously been uploaded to the hosting environment it will not be used and the object from the local environment will be used. This avoids issues relating to different versions of the same class or the possibility of two different objects having the same name.

Each hosted object is associated with a classloader, which is used to maintain a separation between hosted objects and also allows a fine grain control on which objects are to be unloaded. Unloading objects is a two stage process, each hosted object has a soft and hard time to live. Each time to live is a wall clock time and each call made to a method within the object resets the clock. The ttls are expressed as a time to keep the object alive for, the soft limit is when to cache the object to disk and remove it from memory, the hard limit is when to delete the object from the hosting environment. For example if we have ttls of 5 and 10 minutes for the soft and hard limits respectively, after five minutes of inactivity the object will be cached to disk then after a further 10 minutes the object will be deleted. Up until this point any object reference which pointed to the hosted object on this server would still resolve to a value object, but after deletion the object and associated state is removed from the system.

4.2 Creating Distributed Applications

SWIM provides several forms of distributions, all of which can play a part in successfully distributing an application. The ways of distributing an application within SWIM are a trade-off between power provided to the programmer and ease of creating a distributed application. Firstly SWIM can be used in the traditional role where the application is built using the middleware, as in the case of RMI, Corba and Web Services. Secondly it provides support for distributing application which were not designed with distribution in mind. This could be done for performance or reliability reasons. Spreading a compute intensive application over several computers allows for a potential increase in performance. SWIM supports object replication, where a single object is copied into multiple locations on the network. These replicated objects can then be accessed as if they are a single object within the client application, if the connection to a single copy of the object is lost then the application transparently continues with the remaining copies of the object.

SWIM primary design concerns are flexibility and ease of use. It provides three forms of interaction to the programmer. Firstly SWIM can be used as a traditional object based middleware, such as RMI or CORBA, where the application is designed and built with distribution in mind. This series of APIs is the lowest level and gives the programmer full control on all aspects of the middleware. With these APIs the programmer can remote arbirtary objects and to clone and migrate objects during the application runtime.

Secondly, SWIM can be used to manually transform an application to be distributed. In this case the application is written without consideration to distribution and with no specific middlware in mind. When the application is to be distributed there are two ways in which SWIM and interact with the application. If the programmer has access to the source code of the program SWIM can be made part of the application using the same APIs as before.

If the source code is not available for modification then SWIM allows the programmer to provide a configuration file to the SWIM application loader. The SWIM application loader uses reflection and introspection to transform an application for use in a distributed environment. It takes a simple configuration file similar to an access control list. The configuration file consists of a list of classes to which are to be hosted within the SWIM pool and a list of classes which are explicitly not to be hosted within the SWIM pool. By default SWIM will host any class and subclass of a any class which appears in the 'to remote' list, this can be overridden by using the 'not to remote' list. The combination of the two lists provides fine grained control of which classes are to be hosted within the SWIM pool and which classes should remain local. Each control list allows both absolute class names or just package names. By default all the SWIM classes and the standard JDK classes are denied being hosted in the SWIM pool

Finally SWIM can transparently distribute an application with no programmer intervention. Although this provides the simplest method for application distribution it comes at a price. All objects, except the SWIM and standard java classes are available to be hosted with the SWIM pool.

The flexibility of SWIM means that it is possible to mix all of these forms within a single application, allowing the programmer to move from automatic distribution, through manual augmentation to a designing and implementing aspects of the application manually. The programmer is able to choose which aspects of the application are worth paying attention to and which can be left to SWIM.

4.3 Reliability

SWIM Objects may be clustered to increase the reliability of the system. There are three forms of clustering available. Firstly, parallel clustering where a number of remote objects is bound together and referenced as a single object. The result from the first remote object to return from the method call is used as the result and is returned to the calling method.

Secondly, a number of remote objects is bound together but a vote is held on what the return result should be, once the majority of remote object have returned the same value the returned value is returned to the local context.

In both of these cases any remote objects which cannot be contacted are removed from the pool of clustered objects. Additional objects can be created and added to the cluster to replace any remote object which is no longer contactable.

The final form is no clustering. This is for each local reference to and object there is a single remote object.

5 Testing and Evaluation

SWIM was evaluated against two other middleware technologies, Apache's XML-RPC and Java RMI. The testing and evaluation of SWIM is two-fold, firstly it investigates the call performance overhead that SWIM entails when used in a number of scenarios ranging from simple methods with no parameters to parameters which take a custom datatype.

Finally SWIM is used to distribute a simple application across a pool of machines.

We evaluate SWIM on two accounts firstly there is the time to perform a remote call and secondly on the amount of impacts SWIM has had on the design on the system.

5.1 Testbed

The testbed is composed of 20 nodes, each is a Pentium 4 3.0 Ghz with 1 Gigabyte of memory connected via a 1 Gbit/s switch. Each node runs Linux kernel 2.6.9-34 with Java 1.6.

We examine SWIM's per method call performance in comparison to other middleware technologies. For this experiment we used three machines from our testbed pool. Each of the three machines ran one of: the server process, the client process and the finally a registry.

RMI and XML-RPC were used to provide a baseline comparison. We used Apache XML-RPC version 3.1[4] and the version of RMI bundled with JDK 1.6. The interface shown in Listing 1 was implemented in Apache XML-RPC, RMI and SWIM.

```
public interface TestCases {
public int noParameters ();
public int primitiveParameters (
        boolean aBoolean,
        int anInteger,
        double aDouble);
public int objectParameters (
        Boolean aBoolean,
        Integer anInteger,
        Double aDouble);
public int complexParamaters (
        ComplexType ct);
}

public class ComplexType {
String aString;
int anInt;
double aDouble;
byte[] byteArray;
}
```

Listing 1. Interface used for performance evaluation

[4]http://ws.apache.org/xmlrpc/

We examine the call performance of RMI, XML-RPC and SWIM using the method interfaces shown in Figure **??**. Each of the methods was implemented to return 0 so that an evaluation of the communications overhead, without any processing overhead could be made.

The first test uses a remote method which takes no parameters. The second method takes 3 primitive types (a boolean, and integer and a double), the third method takes the same three types as objects, for the int is passed as an Integer. The final method takes a user defined object, of class ComplexType. The string parameter was 25 characters long and the byte array was set to 128 bytes. In each case method call was made 100000 times and the minimum, mean and maximum call times are reported.

	XML-RPC	SWIM	RMI
Lowest	1.45	0.57	0.1
Mean	1.68	0.71	0.12
Highest	204.11	5.42	5.65

Table 1. No parameters calls times in ms

	XML-RPC	SWIM	RMI
Lowest	1.54	0.69	0.11
Mean	1.79	0.82	0.12
Highest	43.22	3.58	1.95

Table 2. Primitive types call times in ms

SWIM is around seven times slower than RMI for method with no parameter (Figure 1), or with primitive types (Figure 2) for the parameter. This gap reduces to around four time slower for objects from the JDK (Figure

	XML-RPC	SWIM	RMI
Lowest	1.53	0.72	0.19
Mean	1.78	0.84	0.20
Highest	43.90	2.30	10.27

Table 3. Object parameters call times in ms

	XML-RPC	SWIM	RMI
Lowest	1.64	0.86	0.19
Mean	1.94	0.97	0.20
Highest	201.98	3.85	6.01

Table 4. Complex Types call times in ms

3) and user defined object (Figure 4). In all cases SWIM is around twice as fast as Apache's XML-RPC.

5.2 Distributing an example application

We created a simple application which ran on a single machine. The application simulated a process heavy application. We had a single class which had one method $public$**count**() which counts from -2^{31} to $2^{31} - 1$ in increments of one. The test application creates twenty of threads with each thread running the count method. The application was run 50 times and the mean and 95% confidence interval were calculated and are shown on in figure 3. The number of SWIM nodes was 0, 2, 4, 8 and 16. In the case of zero SWIM nodes the SWIM middleware was not used an the application is running locally, in all other cases the SWIM middleware was used and all invocations of the count method are remote.

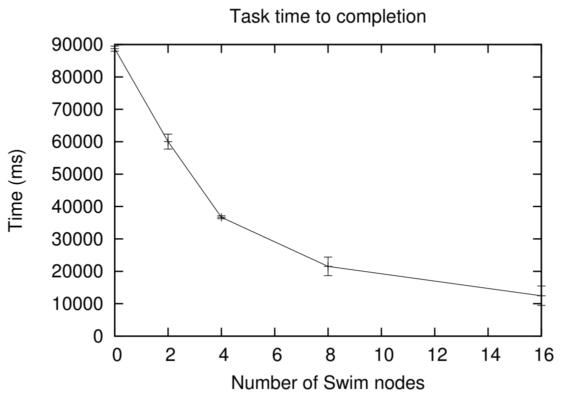


Figure 3. Task completion times vs number of SWIM nodes

Figure 3 shows the mean time to complete the compute task over 50 runs. The application was unmodified in all cases and the only change was the size of the SWIM pool. Figure 3 shows that SWIM can be used to increase the processing power available to an application, even if the application has not been written with distribution in mind.

6 Limitations

SWIM does not have a serious form of security and authentication for the SWIM pool. The closest it has are the use of SSL sockets. Although these provide an encrypted communications channel they do not stop any node from making a connection to a SWIM node and attempting to execute methods in any hosted object.

7 Conclusions and Future work

We have presented a motivation for dynamic and flexible middleware which can be used in a number of scenarios and which can easily change as the application evolves. Allowing the programmer to decide how dependent he/she wishes to be on the middleware at any given time.

We have presented SWIM and compared its overheads, both programmer time and remote call time. Although it is slower to execute than RMI the time taken to create a solution using SWIM is much lower than when using RMI. It was faster than XML-RPC in all cases and required significantly lower programmer time. XML-RPC required the implementation of a custom parser and serialiser before it was able to pass ComplexType object from the client to the server. RMI required that the ComplextType object implemented the Serializable interface whereas SWIM imposed no such requirement.

SWIM is rather simplistic when it comes to the automatic distribution of objects with a SWIM pool. Once an application has been distributed with the pool there is no automatic object migration so that objects are located to hosts which are more suitable for them, such as collocating objects which communicate a lot or moving a process intensive object to a SWIM pool node which is lightly loaded.

7.1 Further work

We envision a system which could be used to provide increase processing capabilities to the application programmer with minimal effort on the programmer's part. We wish to remove the complication of dealing with a distributed system from the application programmer whilst still having the advantages of increased processor power to perform scientific calculations. The components of the system can be chosen as run-time and distributed across the available computing resources with the trade-offs of several aspects in mind, mainly performance and reliability.

The requirements of an application are expected to evolve and change over time. As such each aspect of the application can be expected to change, including how the application is distributed. We propose a system which provides a range of APIs to the application programming, ranging from transparent distribution of an application, through guided distribution, to a standard API where objects are designed and locations are assigned. This allows the programmer to make decision concerning how the application is to be distributed only if they wish to do so and only if it would be beneficially to the application. Providing a separation of concerns of the application logic and the distribution logic allows the programmer to concentrate on the application logic.

References

[1] W. Cazzola, A. Ghoneim, and G. Saake. Ramses: a reflection middleware for software evolution. In *n Proc. of the 35th Annual Meeting of the ACL and the 8th Conf. of the EA CL (A CL/EA CL '97*, pages 56–63, 2004.

[2] J. B. Dennis. First version of a data flow prodecure language. In *Programming Symposium: Proceedings, Colloque sur la Programmation (LNCS, vol 19)*, pages 362–376, 1974.

[3] D. Gelernter. Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985.

[4] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. The MIT Press, 1994.

[5] G. Kirby, S. Walker, S. Norcross, and A. Dearle. A methodology for developing and deploying distributed applications. *Component Deployment, ser. Lecture Notes in Computer Science*, 3798:37–51, 2005.

[6] K. T. Lam, Y. Luo, and C.-L. Wang. Adaptive sampling-based profiling techniques for optimizing the distributed jvm runtime. In *2010 IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 1–11, 19th–23rd April 2010.

[7] G. Lee and J. Morris. Dataflow java: Implicitly parallel java. In *Computer Architecture Conference, 2000, ACAC 2000*, pages 42–50, 31 Jan - 03 Feb 2000.

[8] A. Spiegel. *Automatic Distribution of Object-Oriented Programs*. PhD thesis, FB Mathematik und Informatik, Freie Universitat Berlin, August 2002.

[9] G. K. Thiruvathukal, P. M. Dickens, and S. Bhatti. Java on networks of workstations (javanow): A parallel computing framework inspired by linda and the message passing interface (mpi). In *Concurrency: Practice and Experience Special Issue: Message passing interface-based parallel programming with Java*, volume 12, pages 1093–1116, 2000.

[10] I. Wang. P2PS (Peer-to-Peer Simplified). In *Proceedings of 13th Annual Mardi Gras Conference - Frontiers of Grid Applications and Technologies*, pages 54–59. Louisiana State University, February 2005.

[11] W. Zhu, C.-L. Wang, and F. C. M. Lau. Lightweight transparent java thread migration for distributed jvm. In *International Conference on Parallel Processing*, pages 465–472. IEEE, 2003.