

Java and Distribution of Applications Requiring Mutual Exclusion and Deadlock Detection

Judith Bishop¹, Basil Worrall¹, Karen Renaud², Johnny Lo¹,

¹ Department of Computer Science, University of Pretoria. {jbishop,bworrall,jlo}@cs.up.ac.za

² Department of Computing Science, University of Glasgow. karen@dcs.gla.ac.uk

ABSTRACT

Programming in a distributed environment is a complex activity. Programmers need to be aware of issues unrelated to their domain of expertise, and are often unprepared for the challenges that distribution brings. Chief among these are issues of mutual exclusion and deadlock detection, which have quite different solutions in a distributed environment, compared to a centralised one, as exhibited on an ordinary workstation. We have tackled this problem by adopting the technique of separation of concerns. We have developed a framework called Algon which enables complex distributed algorithms to be incorporated into existing or new distributed systems as required. For each algorithm class, a scheduler is defined that handles the communication between distributed sites, and can be instantiated together with any one of a range of suitable algorithms that solve the problem. Algon therefore frees ordinary programmers from needing to know the intricacies of complex distributed algorithms.

This paper concentrates on revealing how Java has contributed to the successful implementation of such a framework. As well as the inherent object-oriented nature of Java, we have made extensive use of RMI, properties and dynamic class instantiation and synchronization across remote sites. The example chosen to illustrate these features in this paper is distributed deadlock detection. Nevertheless, there is considerable similarity with the mutual exclusion class of algorithms reported on earlier. The paper concludes with future plans for Algon, chiefly in the area of performance improvement. **Keywords:** Framework, distributed algorithms, distributed systems, separation of concerns, autonomy, performance, tailoring.

1. INTRODUCTION

A distributed system can be defined as *a system consisting of several computers that communicate across a network*. These computers do not share memory or processor resources. *Distribution* is the term given to the program-

ming paradigm used when developing applications for such distributed systems, where the objective is to harness the computational strength of the entire system. The Java programming language, and its use in distributed systems, is the focus of this paper, mainly due to its popularity and suitability for the development of distributed applications.

Java for scientific usage focuses primarily on parallel, network and internet applications [20]. The goals and challenges of this work centre around performance and communication mechanisms [11, 8, 16], yet developing applications in such an environment is a complex activity [10].

Many programmers have been trained to develop on workstations and find the complexity of the distributed paradigm hard to handle. In addition to the normal cognitive and abstract nature of the programming activity itself, concerns in a distributed environment include, for example, non-determinism, contention and synchronisation issues [14]. Programmers may be faced with a need to guarantee distributed mutual exclusion, or to achieve distributed termination, or detect deadlock, for example. There already exists a rich base of research from the 1980s and 1990s for solving such problems. A range of algorithms has been classified according to their function and each algorithm achieves the expected result in a different way and with different performance characteristics. The exact implementation of the algorithms in a particular programming language is often left unspecified [24, 25]. The programmer is therefore faced with

1. deciding which algorithm is best to use, and
2. actually implementing the algorithm in a distributed fashion in Java.

While many aspects of programming distributed applications are challenging, one of the essential issues that many programmers find most difficult to deal with is the incorporation of these distributed algorithms into their systems. It may be necessary to do this if there is a need to guarantee distributed mutual exclusion, or to achieve distributed termination, for example. These distributed algorithms tend to be difficult to understand, and harder to implement properly. Programmers are often unable to evaluate them critically and therefore cannot make an informed decision about the correct algorithm to use. For example, one of the sim-

plest algorithms for guaranteeing mutual exclusion, Ricart-Agrawala [23], involves

1. sites sending requests and replies to other participating sites,
2. comparing timestamps, and
3. keeping queues of waiting sites.

Other algorithms providing better performance have even greater complexity. Although systems exist for illustrating and comparing the functioning of algorithms [1, 3, 13], their primary function is educational and they are not intended to be components for development.

Rather than trying to educate the vast number of programmers involved in developing distributed systems about the functioning, performance and complexity of various algorithms, we propose that the *separation of concerns* technique be applied [21, 15]. Separation of concerns simplifies the programmer's task by enabling him or her to deal with various aspects of the programming process *separately*. Programmers can then concentrate on specific tasks individually, and remove difficult and complex tasks from their realm of responsibility and control. Separation of concerns also allows the programmer to decompose software into smaller, more manageable parts that are easier to keep up to date with evolving needs [18]. This technique has been applied to other aspects of distributed applications [9], and to separating algorithms from parallelism[26] but not as yet to distributed algorithms.

An important aspect of the separation of concerns ethic is providing mechanisms to implement synchronization code separately from any code to which it may apply. Synchronisation is usually done by means of the use of a critical section and some synchronisation mechanism such as semaphores. When one needs to achieve synchronisation in a distributed system one needs to make use of a distributed algorithm, and semaphores are simply no longer available, as there is no shared memory. We propose that complexity related to distributed algorithmics, such as synchronisation, be hidden from the programmer in its own component level. This is achieved within a framework called Algon¹.

Algon includes:

- a *library* of algorithms to be used as and when required;
- a *framework* for incorporating an algorithm into the system;
- a *tool* for evaluating different algorithms based on their performance within the distributed application;

Many programming languages routinely provide libraries of mathematical functions to simplify the programmer's task. Algon provides programmers with a variety of distributed algorithms so that they can experiment and thereby select

¹Algon stands for *Algorithms On the Net*.

the algorithm with the best performance for their particular application.

Algon is implemented entirely in Java and, as such, provides Java classes which can be incorporated into new or existing programs. The Algon concept and its associated design pattern was first proposed in [2]. It has since been successfully implemented, and the performance tool has been developed [22]. The purpose of this paper is primarily to discuss the influence of Java in the implementation of the Algon framework, along with details of its use for mutual exclusion and to introduce a further class of algorithms – deadlock detection. Several advanced features of the Java programming language were used, which contributed to the success of the ideal of separation of concerns.

The prime motivation for the existence of distributed algorithms is that the failure semantics of distributed systems is different from those of centralised systems. We do not directly address issues related to these semantics in this paper since they will be dealt with in a later phase of the project. For the purpose of this study we have assumed that the failure semantics of algorithms in a specific family are similar, and that faults are reported by means of reported exceptions thrown by the algorithm processes.

Section 2 presents a overview of the architecture and rationale of Algon. Section 3 considers the Java features which made the implementation of the Algon framework possible. Section 4 gives an overview of an extension into a new category of algorithm. Section 5 discusses plans for future work, and Section 6 concludes.

2. THE ALGON CONCEPT

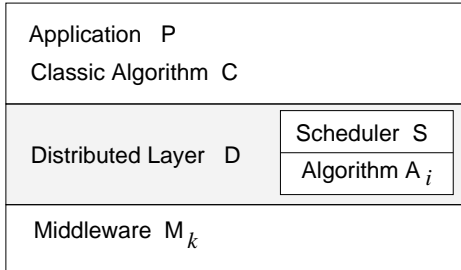
Middleware frameworks have been shown to be viable for the deployment of software components on the Internet [5]. However, they tend to require a certain amount of tailoring of the application and the introduction of new methods. The problem with the wide-scale deployment of components over the Internet is that the frameworks and wrapper interfaces each have to be custom-made for a given system. This is wasteful in terms of time and money and is doomed to failure if the programmers do not understand the functioning of the legacy software.

Another problem is the fact that programmers may not sufficiently understand the nature and needs of the distributed system itself. An application may need to employ distributed mutual-exclusion, snapshots or deadlock-detection algorithms but the algorithms in common use are often centralised and inappropriate for distributed systems [24].

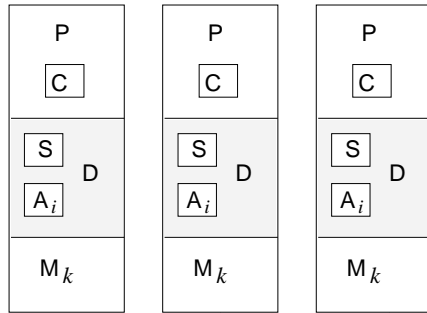
The objective of Algon is to produce a library of pre-coded distributed algorithms which can be incorporated at Java class level into a distributed component-based application while maintaining a clear separation between components. Having decided to treat distributed algorithms as a candidate separate concern, the next step is develop a mechanism for structuring the base code so as to enable this separation [17]. Algon's algorithms have been implemented in such a way that the code is split into logical processes, enveloped within a distributed framework. The resulting component can then be added to a distributed system with the mini-

mum of changes to the original code of the application.

To illustrate the use of Algon, consider the situation where there is a need to distribute the behaviour previously provided by the algorithm C . The programmer would traditionally have to recode the algorithm and incorporate it into the application *in place of* C [7]. Algon can provide a mechanism whereby distribution features can be *added* to the application in the form of a component.



(a) Abstract Framework



(b) Example on three nodes

Figure 1: Distributed View of the Application using an Algon component.

The proposed architecture for application systems is shown in Figure 1(a). The application-specific code, P , and classic algorithm, C , remain unchanged. Figure 1(b) illustrates how this architecture might be implemented on a system with three nodes. In order to distribute C 's behaviour, the system is extended by adding:

1. a distribution layer, D , which consists of:
 - (a) a scheduler S , and
 - (b) an algorithm A_i .

The distribution layer, D , is selected specifically to match the algorithm, C . In addition, the distribution layer will usually need a *request set* which identifies the nodes in the system.

2. a middleware backbone M_k , which facilitates communication with other participants. It can use any suitable communication structure such as Java RMI, CORBA IIOP, DCOM or .NET.

Our intention is to provide developers with a range of distributed algorithms for any particular problem. To make these algorithms easily interchangeable, a standard interface is implemented for specific types of algorithms. For a specific classic problem i , the interface I_i is used by the scheduler

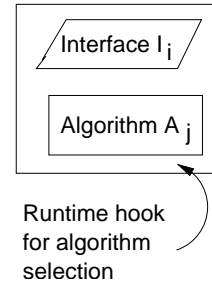


Figure 2: Abstract Framework

to interact with all algorithms implementing that interface. This makes it easier to introduce new algorithms and to specify, at runtime, the algorithm that should be used. Figure 2 shows the algorithm interface I_i being implemented by an algorithm A_j . An example of this is the MENT (Mutual-Exclusion Non-Token Group of Algorithms) interface being implemented by the Ricart-Agrawala mutual exclusion algorithm [24]. Another example of this is the DD (Deadlock-Detection Group of Algorithms) interface being implemented by the Chandy-Misra OR model deadlock-detection algorithm [4] (Figure 3).

The developer can choose the algorithm to be used either by means of a graphical user interface or by directly editing a configuration file. The developer also indicates the identities of the nodes involved in the distributed system. This makes the algorithms interchangeable without any need for recompilation.

3. JAVA'S ENABLING ROLE

This section explains how Java enabled the successful development of the Algon framework. Section 3.1 will discuss the use of RMI as Algon's middleware backbone. Section 3.2 will explain how the algorithm is chosen by the developer and plugged into the system at runtime. Section 3.3 will discuss the use of the Java synchronization mechanism in protecting key resources at each site.

3.1 Java RMI

An essential consideration when developing a middleware framework is in deciding the type of middleware backbone to be used. The advance in the state-of-the-art in this field means that there is a variety of options available, and frequently the middleware can free the programmer from language-dependence. The use of a *Remote Procedure Call* (RPC) architecture seems the natural choice for developing a middleware framework, because of its ease of use and general elegance. Consequently, the Java RMI architecture was chosen for the Algon framework. As shown in Figure 1(b), the algorithm instances communicate via this medium. In order to achieve a true *remote procedure call*, one site first needs to establish a remote interface, from which *stubs* can be generated. Stubs act as local proxies for remote objects

so that the local object invokes methods on the stub as if the remote object were at the same site. The stub performs the low-level serialisation of data and communicate the data back and forth between the caller and the callee in a transparent manner. A typical remote interface as used in the Algon framework is given below. This interface represents mutual exclusion non-token passing (i.e. Ment) algorithms.

```
public interface Ment extends Remote {
    // Used by the Scheduler
    public void sendRequests(SchedulerInterface si)
        throws RemoteException;
    public void getRequestSet()
        throws RemoteException;
    public void sendRelease() throws RemoteException;

    // Used by other Ment algorithms
    public void reply() throws RemoteException;
    public void request(long time,
        SchedulerInterface si, Ment m)
        throws RemoteException;
    public void release() throws RemoteException;
}
```

This example represents the interface advertised by a distributed mutual exclusion algorithm containing methods to be invoked by a caller. Each method defined in such a remote interface must be declared as throwing a `RemoteException`, as the potential exists for some unexpected event to filter back to the caller.

It is worth noting is that only three of the methods in the interface given above, i.e. `reply`, `request` and `release`, are used in a distributed setting. The remaining methods will be used by the Scheduler to initiate local procedure calls between it and its own instance of an algorithm.

Any program wishing to invoke methods on objects not in its own namespace needs some form of reference to that object. Most RPC architectures use a naming service to achieve resource discovery. The naming service discovers resources and acquires references to remote objects. Once objects have references to one another they can communicate. We illustrate the binding process, initiated by the Scheduler using the next block of code.

```
private Ment algorithm;
public Scheduler(String id) {
    .
    .
    algorithm = ... // instantiate the algorithm
    try {
        Naming.rebind("rmi://" + id, algorithm);
        algorithm.getRequestSet();
    } catch (Exception ex) {
        ex.printStackTrace();
    }
}
```

By this means an instance of the chosen algorithm is created, and then bound into a naming service (we will return

to a discussion regarding exactly how a *specific* algorithm is chosen in Section 3.2). As soon as the algorithm instance is bound a *request set* needs to be set up. In terms of distributed mutual exclusion algorithms, a *request set* is the set of sites which may hold the lock for the critical section at any given time, and therefore the sites need to be queried before another site may proceed into the critical section. The manner in which these queries and responses are treated, and the physical nature of the request set, defines the difference between different algorithms. To set up the request set we invoke the `getRequestSet()` method on the local algorithm.

However, there is a snag. There is a difficulty in establishing exactly *when* an object should attempt to physically obtain a reference to another, given that the required reference will not exist in the naming service registry until an object had been bound to it. This is not a trivial problem, since the participating sites may be running at different speeds and certainly will take different amounts of time to bind themselves to their local naming service.

We solved this problem by applying the following strategy:

1. Through some means, provide a list of the objects that should be available to a participating site for remote invocation purposes. Actual references can, of course, not be supplied, but it *is* possible to provide a unique name for each participating algorithmic object (the name by which the object is to be bound in the naming service registry).
2. Each object will then continuously query the naming service for the names it has been supplied with, until it has a reference for each name.

It can be argued that this strategy is extremely rigid, and that a more dynamic approach to resource discovery is desirable. However, for systems with known boundaries and behaviours, it is adequate. The source code implementing the strategy is given in the `getRequestSet()` method below, which provides some insight into the functioning of the system's initialisation process:

```
// getRequestSet() method
Object o = null; while (o == null) {
    try {
        o = Naming.lookup("rmi://" + uniqueName);
        requestSet.add(o);
    } catch (Exception cioe) {
        o = null;
        delay(1000);
    }
}
```

3.2 Properties and Algorithm Instantiation

Since our framework is designed to provide a library of algorithms, it is important to address the question of how a different algorithm is chosen for the application. Two techniques are applied to address this. Firstly, the `Properties` class, which can be found in the `java.util` package, is used

to gain information about the framework administrator's choice of algorithm, and about the physical structure of the distributed system. A special properties file is used to allow the developer to record information to be communicated to the Algon layer at runtime. This properties file can be edited either manually or by means of a graphical user interface. The following two blocks of code illustrate how the properties are used.

- From the Scheduler:

```
FileInputStream fin = null;
try { //locate the properties file
    fin = new FileInputStream("scheduler.prop");
} catch (FileNotFoundException fnfe) {
    System.exit(1);
}

Properties props = new Properties();
try { // load the runtime
properties
    props.load(fin);
} catch (IOException ioe) {
    System.exit(1);
}

String algName = props.getProperty("Algorithm");
if (algName == null) {
    System.exit(1);
}
```

- From an algorithm implementation:

```
FileInputStream fin = null;
try {
    fin = new FileInputStream("hosts.prop");
} catch (FileNotFoundException fnfe) {
    System.exit(1);
}
Properties props = new Properties();
try {
    props.load(fin);
} catch (IOException ioe) {
    System.exit(1);
}

String temp = props.getProperty("numHosts");
if (temp == null) {
    System.exit(1);
}
int numHosts = Integer.parseInt(temp);
for (int i = 0; i < numHosts; i++) {
    String name = props.getProperty("host" + i);
    int numAlgs = Integer.parseInt(
        props.getProperty("host" + i + "Algs"));
    for (int j = 0; j < numAlgs; j++) {
        String uniqueName =
            props.getProperty("host" + i + "Alg" + j);
        Object o = null;
        while (o == null) {
            try {
                o = Naming.lookup("rmi:///" + uniqueName);
```

```
                requestSet.add(o);
            } catch (Exception cioe) {
                o = null;
                delay(1000);
            }
        }
    }
}
```

The code from the algorithm implementation shows how a unique identity for each algorithm instance is found (unique in terms of IP address and registry name). Having solved the problem of establishing the name of the algorithm to load (see the code from the Scheduler), the instantiation of the algorithm is relatively simple. This forms the application of the second technology mentioned above. Passing the fully qualified name of the class to the static `Class.forName` method, an instance of the class `Class` is generated. It is now possible to generate an instance of the class, using the `Class` object's `newInstance` method (which can take constructor parameters, if need be), and to type-cast the instance (which is `Object` by default) to the remote interface type. All this is demonstrated in the following five lines of code.

```
try {
    algorithm =
        (Ment)(Class.forName(algName).newInstance());
} catch (Exception ex) {
    ex.printStackTrace();
}
```

Both `forName` and `newInstance` throw exceptions, which flag that the class may not be available, or that some corruption of the properties file has occurred.

This section started by discussing the remote procedure call architecture and how it is implemented in Algon. We have also covered the discovery of the specific algorithm required for a particular application execution. This section completed this part of the discussion by explaining how this information can be used to instantiate the algorithm and connect to remote instances of the algorithm.

3.3 Synchronization

The final aspect of the Java implementation of Algon is the use of the synchronization mechanism of the Java in Algon. The synchronization of threads and processes on a single PC (with or without multiple processors) is a field of study all on its own, and many methods have been proposed and implemented to deal with the concurrency issues such systems pose. However, when concurrency is applied in a distributed context, the approaches are not all that clear cut. Fortunately, the use of monitors in the Java language greatly simplifies the complexity of our solution to this problem. In a nutshell, if a group of instructions needs to run sequentially without avoidable interruptions, they can be enclosed in a `synchronized` block. When a `synchronized` block is encountered at run-time, the monitor of a specified object (usually that of the object within which the code is executing) is obtained. This means that no other `synchronized`

block of code from the same object may execute until the end of the current block.

The use of synchronization in our framework is restricted to the algorithms themselves. It is usually the case that some state information is kept at the algorithmic level, and in a distributed environment it is difficult to keep track of when and how the changes to these state variables are made and so great care must be taken. This is the ideal situation for making use of the Java synchronization mechanism. A good example of the use of this mechanism in an Algon algorithm is demonstrated by the implementation of the mutual exclusion algorithm mentioned in Section 3. When this algorithm sends requests to neighbouring sites, it counts how many request messages were sent, and how many have been replied to (no reply means that another site is in the critical section). This seems to be a minor concern, but, indeed, the correct operation of the algorithm revolves around the correctness of the `numRequests` counter variable. It is therefore critical that this counter be protected as much as possible from interference. Below is the implementation, which illustrates the use of the synchronization block:

```
public void sendRequests(SchedulerInterface si)
    throws RemoteException {
    .
    .
    .
    numRequests = requestSet.size();
    for (int i = 0; i < requestSet.size(); i++) {
        Ment ra = (Ment)requestSet.get(i);
        ra.request(requestTime, si, this);
    }
    while(numRequests != 0) {
        // wait a bit
    }
    // Access to critical section granted
}

public synchronized void reply() +
    throws RemoteException {
    numRequests--;
}
```

When the `sendRequest()` method is called by the Scheduler, the algorithm will send request messages to all of the algorithms in its request set. It will then sit doing nothing until the `numRequests` integer is equal to zero. When an algorithm receives a request message, it responds either immediately by calling the `reply` method on the reference it was passed (see the call to `ra.request` above), or it will queue the request until it has released the critical section it already has. As soon as it releases the critical section, it will call the `reply` method of each of the algorithm instances that sent requests. The `reply` method decrements the `numRequests` integer, so the `sendRequests` method will eventually terminate, and allow the calling Scheduler access to the critical section.

4. APPLICATION OF ALGON

In previous work [2] details of a mutual exclusion algorithm's calling patterns were discussed. As evident in the discussion

on Figure 1, in Section 2, there are other kinds of algorithms. We have also investigated and implemented a distributed layer for deadlock detection algorithms. In this section we discuss how the elements of this layer are set up and how robust the Algon framework proves to be.

4.1 Distributed Deadlock Detection

Many algorithms exist in this category in the literature. According to Knapp [12], the Chandy-Misra edge-chasing algorithms perform well, provide correctness proofs and do not report false deadlocks. Chandy *et al.* [4] have developed two types of edge-chasing algorithms, namely the *AND* and the *OR* models. For the purpose of this discussion we will use the *OR* algorithm, *CMO*, applying it to the classic Dining Philosophers' problem.

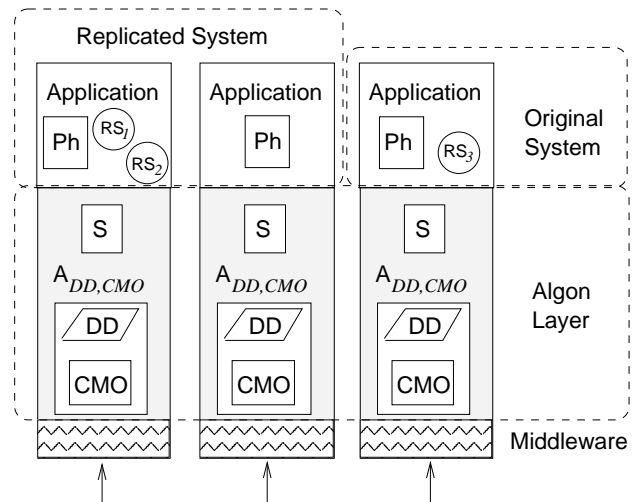


Figure 3: Using Algon to add a distributed deadlock detection algorithm

Figure 3 illustrates how a distributed deadlock-detection algorithm is incorporated into a system with three nodes. Each node has a philosopher, *Ph*, and zero or more optional resources, *RS_i*. The class diagram is shown in Figure 4. There is an apparent contradiction between our assertion of separation-of-concerns and the upward link from the Scheduler to the Philosopher. This seems to break the required separation. However, it must be borne in mind that any potentially deadlock-detection algorithm runs independently of application programs in the system and needs access to the deadlocked processes in order to determine whether a deadlock exists or not. The only way to facilitate this detection process is by checking whether the processes, in this case Philosophers, are possibly inextricably involved in a deadlock loop.

The deadlock-detection activity can be triggered in different ways. The system developer could decide that the application should trigger it, if it has been blocked waiting for a resource for a certain amount of time. This does not necessarily indicate the presence of a deadlock though; it could be that the network is particularly busy. Even if this approach is followed the programmer would have to ensure that the waiting time before triggering the process is adjusted to the current average waiting time. Deadlock detection could also

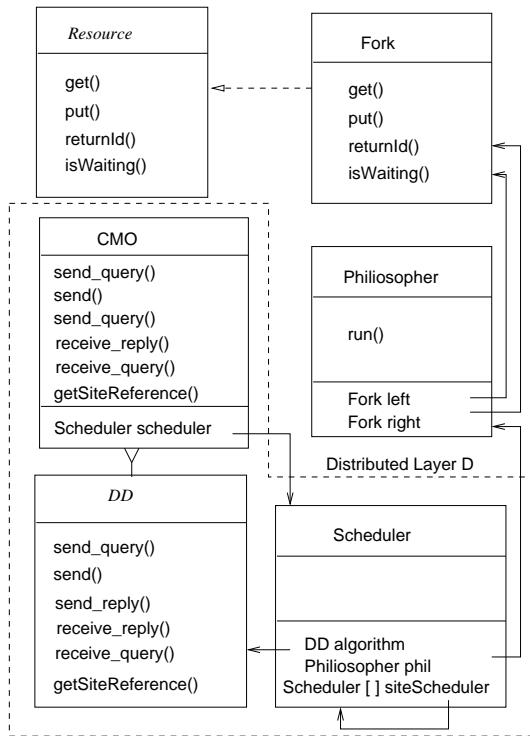


Figure 4: Class Diagram

be done at regular intervals but this tends to slow the system down, and is a waste of resources if deadlocks are infrequent. In order to test our system deadlock-detection was triggered from the test application. The deadlock-detection policy in an industrial setting would be determined by the system developers once the possibility of deadlocks had been ascertained. The Philosopher's execution is shown below:

```
public void run() {
    try {
        think();
        right.get(identity);
        left.get(identity);
        eat();
        right.put(identity);
        left.put(identity);
    }
}
```

Once again this code is recognisable as that which would appear in a centralised system.

The interaction between the different participants in setting up the communication between participating nodes is shown in Figure 5. The resources are assumed to be available before the application starts executing. The setting-up process involves the application instantiating the classic algorithm, in this case the **Philosopher**, and then instantiating the **Scheduler**. All other details of the setting-up process are dealt with inside the **Scheduler**.

Unlike other applications, such as the Readers-Writers prob-

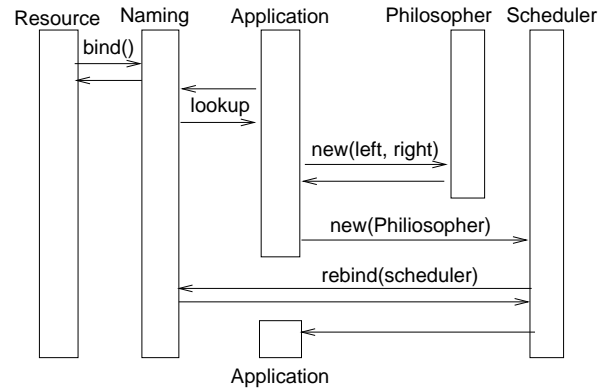


Figure 5: Setting Up with Deadlock Detection

lem which involves the distributed algorithm at each call [2], there is no need for the algorithm to be involved in the calls the **Philosopher** makes to **get** and **put** the **Forks**. The deadlock algorithm is only invoked when the current situation triggers an investigation into a possible deadlock. The

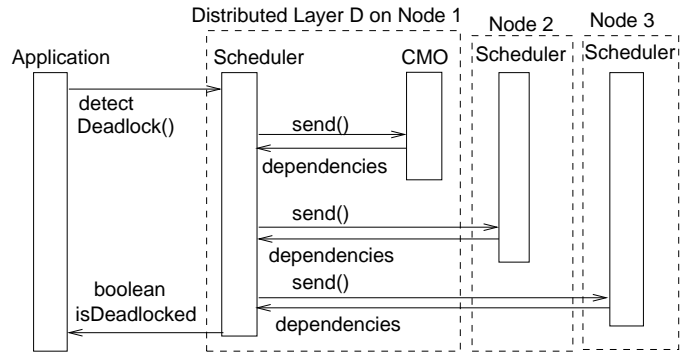


Figure 6: Testing for Deadlock

interaction precipitated when a test for deadlock is triggered is shown in Figure 6. The application, or some other trigger, invokes the **detectDeadlock** method on the **Scheduler**. The **Scheduler** then uses the algorithm on the current node, and **Schedulers** on other participating nodes, to generate sets of dependencies. The situation is analysed and a diagnosis made of the deadlock status of the system. The process triggering the detection process will then have to decide on a suitable response should a deadlock be detected. The Chandy-Misra algorithm detects, but does not *resolve*, deadlocks. A deadlock-resolution algorithm would have to be separately incorporated into the system in order to handle detected deadlocks.

4.2 Assessment

Looking at the implementation of the deadlock detection algorithm's distribution layer, it is interesting to note the points of similarity to the mutual exclusion algorithm. Once again, Java RMI is used as the middleware layer. Communication occurs between instances of the scheduler in this case, using a remote interface. The algorithm in use is specified in the same way (using the **Properties** class), and instantiated using the same **forClass** construct. In this case, no synchronization is needed within the algorithm, although the

Dining Philosophers problem does make use of synchronization to ensure that the concurrency aspect of the problem does not fail.

Therefore we can conclude that the architecture designed for use in the mutual exclusion case has survived relatively intact when applied to a completely different genre of algorithm. In fact, ongoing development with the Algon framework is showing that the architecture is sufficiently scalable to handle several other mutual exclusion and deadlock detection algorithms.

5. FUTURE WORK

The planned development of Algon is shown in Table 1.

In addition to incorporating the basic required functionality, there are also some issues, as yet unsolved, which we will address as future work during the succeeding phases of Algon's development. The following work will be undertaken during the second phase:

1. It is clear from Figure 1 that the entire Algon structure needs to be duplicated for each application in the system. Some work should also be done in determining whether the distribution layer, D , can be shared amongst applications on the same machine. Sharing the scheduler between different processes in the deadlock detection scenario is trivial.
2. We intend finding points of similarity between the schedulers tailored for certain algorithms, so as to create a Scheduler Interface that can be used to specify a generic behaviour expected of all Algon schedulers.

The experiences of the second phase will probably suggest a number of new directions for the third phase, but at present the following are envisaged:

1. Different algorithms in a system may need to interact with one another. An example of this is the way the deadlock-detection and deadlock-resolution algorithms need to work in order to deal with deadlock situations. This handling of interacting concerns needs to be investigated [19].
2. Distributed systems enhance fault-tolerance in some ways [10]. They increase the possibility of replication which in turn enhances reliability. However, it should be borne in mind that the failure semantics of any distributed system is radically different from those of a centralised one. The user will never be oblivious to the true nature of the system because sooner or later an error will be reported which relates to the distribution of the system. The inclusion of Algon also introduces a new range of possible errors. This interferes with the true separation we are trying to achieve, but is one of the issues that has not enjoyed any attention thus far. We need to decide on a policy for dealing with such exceptions as are thrown by both the Algon and the middleware layers in order to achieve a certain measure of fault tolerance.

The application of Algon is also being investigated in a Geographical Information System (GIS) system with industrial partners [6]. The intention is to maintain an already-stable architecture once it is distributed on the Internet.

6. CONCLUSION

We have proposed a way of adding distributed algorithms to applications in such a way that the algorithm can be pre-coded by professionals and used by developers without undue concern as to the intricacies of the algorithm. The approach is characterised by reliance on interchangeable components at defined levels thus achieving separation of concerns. We have successfully applied this approach to the classic Readers-Writers and Dining-Philosophers problem and are pursuing the viability of the approach by applying it to other classic problems. We have evaluated the results of Algon's first phase according to well-known separation-of-concerns criteria and find that it performs well.

Acknowledgment

This work was partially funded by grants NRF194 and NRF2969 from the National Research Foundation of South Africa. Our thanks to Peter Dickman for his valuable comments on an earlier draft of this paper.

7. REFERENCES

- [1] M. Ben-Ari. Interactive Execution of Distributed Algorithms. *ACM Journal of Educational Resources in Computing*, 1(2es), Summer 2001.
- [2] J. M. Bishop, K. V. Renaud, and B. Worrall. Composition of Distributed Software with Algon — Concepts and Possibilities. In *Workshop on Software Composition. SC 2002.*, Grenoble, France, April 6-14 2002. ETAPS.
- [3] S. Burdette, T. Camp, and B. Bynum. Distributed BACI: A Toolkit for Distributed Applications. *Concurrency and Computation: Practice and Experience*, 12(1):35–52, January 2000.
- [4] K. M. Chandy, J. Misra, and L. M. Haas. Distributed Deadlock Detection. *ACM Transactions on Computer Systems*, 1(2):144–156, May 1983.
- [5] S. M. Coetzee and J. M. Bishop. A New Way to query GIS on the web. *IEEE Computer*, 15(3):31–45, May–June 1998.
- [6] S. M. Coetzee and J. M. Bishop. GIS and the Internet. In *Proceedings of ParCo 2001. MiniSymposium: High Performance GIS: from Parallel Algorithms to Systems*, Naples, Sept. 2001.
- [7] L. Dominick and K. Ostermann. Supporting Extension of Components with new Paradigms. In *OOPSLA 2000 workshop on Advanced Separation of Concerns*, Minneapolis, Minnesota USA, 16 October 2000.
- [8] V. Getov, G. von Laszewski, M. Philippsen, and I. Foster. Multiparadigm communications in Java for grid computing. *Communications of the ACM*, 44(1):118–125, October 2001.
- [9] R. Guerraoui, B. Garbinato, and K. R. Mazouni. Garf: A Tool for Programming Reliable Distributed Applications. *IEEE Concurrency*, 5(4):32–39, Oct./Dec. 1997.
- [10] N. Kaveh and W. Emmerich. Deadlock Detection in Distributed Object Systems. In V. Gruhn, editor, *Joint 8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-9)*, Vienna, Austria, Sept. 2001.
- [11] T. Kielmann, P. Hatcher, L. Borge, and H. Bal. Enabling Java for high-performance computing. *Communications of the ACM*, 44(1):110–117, October 2001.
- [12] E. Knapp. Deadlock Detection in Distributed Databases. *ACM Computing Surveys*, 19(4):303–327, Dec. 1987.

Phase One	Phase Two	Phase Three
<ul style="list-style-type: none"> • Core Structure • Structuring of base code • Coding of algorithms • Configuration mechanism 	<ul style="list-style-type: none"> • Features adapted to User Requirements • Performance Measurement • Testing Framework • Evolution Management • Separation of Scheduler Object and Interface 	<ul style="list-style-type: none"> • Fault Tolerance • Interacting Concerns • Generic Communication Protocol • Separation of Algon & Middleware Layers

Table 1: Algon’s Planned Functionality in the Incremental Phases

- [13] B. Kolehofe, M. Papatriantifilou, and P. Tsigas. Distributed Algorithm Visualisation for Educational Purposes. In *4th SIGCSE/SIGCUE Conference. ITiCSE’99*, Cracow, Poland, June 1999.
- [14] J. Kramer. Distributed Software Engineering. In *16th ICSE Conference*, Sorrento, Italy, May 1994. Invited State of the Art Report.
- [15] C. Lopes and W. Hursch. Separation of concerns. Technical report, College of Computer Science, Northeastern University, Boston, February, 1995.
- [16] J. Moreira, S. Midkiff, M. Gupta, P. Artigas, P. Wu, and G. Almasi. The Ninja Project. *Communications of the ACM*, 44(1):102–109, October 2001.
- [17] G. Murphy, A. Lai, R. J. Walker, and M. P. Robillard. Separating Features in Source Code: An Exploratory Study. In *23rd International Conference on Software Engineering*, Toronto, Canada, May 12–19 2001.
- [18] H. Ossher and P. Tarr. Using Multidimensional Separation of Concerns to (Re)Shape Evolving Software. *Communications of the ACM*, 44(10):43–50, October 2001.
- [19] J. A. D. Pace, F. Trilnik, and M. R. Campo. Building Customizable Middleware using Aspect Oriented Programming. In *OOPSLA Workshop on Separation of Concerns*, Minneapolis, Oct 16 2000.
- [20] C. M. Pancake and C. Lengauer. High-performance Java. *Communications of the ACM*, 44(1):99–101, October 2001.
- [21] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, Dec. 1972.
- [22] K. Renaud, J. Lo, J. Bishop, P. van Zyl, and B. Worrall. Algon: Supporting Inexpert Comparison of Distributed Algorithm Performance. Submitted for publication, June 2002.
- [23] G. Ricart and A. K. Agrawala. An Optimal Algorithm for Mutual Exclusion Algorithms. *Communications of the ACM*, 24(1):9–17, Jan. 1981.
- [24] M. Singhal and N. G. Shivaratri. *Advanced Concepts in Operating Systems*. McGraw Hill, 1994.
- [25] G. Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, 2nd edition, 2000.
- [26] P. Trinder, K. Hammond, H.-W. Loidl, and S. L. Peyton-Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1), Jan. 1998.