Available online at www.sciencedirect.com
SCIENCE DIRECT
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 114 (2005) 65–85
www.elsevier.com/locate/entcs

Algon: From Interchangeable Distributed Algorithms to Interchangeable Middleware

Karen Renaud

Department of Computing Science
University of Glasgow, United Kingdom
Email: karen@dcs.gla.ac.uk

Judith Bishop, Johnny Lo, Basil Worrall

Department of Computer Science
University of Pretoria South Africa
Email: {jbishop,jlo,bworrall}@cs.up.ac.za

Abstract

We present the design and implementation of a framework for interchangeable distributed algorithms. The algorithms are drawn from the set which includes mutual exclusion, deadlock detection and agreement protocols, and we have implemented several examples of the first tonsiswo of these. Algon cts of a library of algorithms, a framework for incorporating them into a new or existing system, and a tool for evaluating comparative performance. In this way, much of the complexity related to distributed systems can be isolated in its own component level and the programmer can choose from among different algorithms in the same class based on performance in a given application. Incorporating many algorithms in a single framework was made possible by the observation that algorithms in a given class, e.g. mutual exclusion, almost always expose the same methods. These methods interface with an Algon scheduler which maintains state and provides convenient hooks for the application to invoke the services of the algorithm.
In this paper we describe the structure of Algon in detail, with a distributed deadlock detection algorithm as the case study. We then extend the notion of separation of concerns by creating an addition layer in Algon, underneath which the middleware that runs on each node can be interchanged, for example from Java-RMI to CORBA. Challenges in the re-tooling of the system, related to multiple inheritance, exceptions and the automatic generation of stubs and skeletons in our implementation language, Java, were overcome in novel ways. Algon has the potential to be a framework with a long life, as it can adapt to new middleware, such as .NET.

Keywords: distributed algorithms, interchangeability, components, middleware independence, separation of concerns, programming framework

doi:10.1016/j.entcs.2004.02.064

1 Introduction

Many programmers have been trained to develop on workstations and find the complexity of the distributed paradigm hard to handle. Middleware technologies hide much of the detail involved in achieving language interoperability and simplify maintenance, but they also introduce easily missed complexities that isolated systems seldom exhibit [7]. In addition to the normal cognitive and abstract nature of the programming activity itself, concerns in a distributed environment include non-determinism, contention and synchronisation issues [12]. Programmers may be faced with a need to guarantee distributed mutual exclusion, to achieve distributed termination, or to 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. However, the exact implementation of the algorithms in a particular programming language is often left unspecified [26,27]. The programmer is therefore faced with

(i) deciding which algorithm is best to use, and

(ii) actually implementing the algorithm in a distributed fashion.

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 required distributed algorithms into their systems. For example, one of the simplest algorithms for guaranteeing mutual exclusion, Ricart-Agrawala [24], involves:

(i) sites sending requests and replies to other participating sites,

(ii) comparing timestamps, and

(iii) 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 [2,11,25], 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 [19,13]. 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 [6], and to separating algorithms from parallelism [29] but not as yet to distributed algorithms.

We propose that as much complexity related to distributed algorithmics as possible be hidden from the programmer in its own component level. This is achieved within a framework called Algon [1], which 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.

Algon provides programmers with a variety of distributed algorithms so that they can experiment and thereby select the algorithm with the best performance for their particular application.

The Algon concept and its associated design pattern was first proposed in [3]. It has since been successfully implemented, and a performance comparison tool has been developed [23]. In this paper we explain how the Algon framework operates, how the programmer can use it to interchange and compare algorithms, and how this concept can be extended so that the existing Java-RMI middleware can be dynamically replaced by CORBA. Algon is implemented entirely in Java, a popular and suitable language for the development of distributed applications. We envisage a C# version of Algon in which .NET or any other future middleware system is used. The lessons learnt through this research include practical aspects of software composition, the impact of language features on system design and issues associated with evaluating complex systems at runtime.

Section 2 describes the architecture and general design rationale of Algon. Section 3 describes how the Algon framework has been used to make different kinds of algorithms available. Section 4 describes the mechanisms used to make the middleware level interchangeable. Section 5 explains how Algon is used to support decisions about algorithm choice. Section 6 considers related work, and Section 7 concludes.

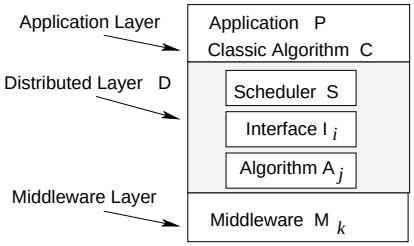

Fig. 1. *Basic Algon Infrastructure*

2 The Algon Concept

The abstract architecture for distributed applications extended with the Algon framework is shown in Figure 1. The application-specific code, P uses an interface to the component, C, using the classic algorithm type, In order to distribute C's behaviour, the system is extended by adding:

(i) a distribution layer, D, which consists of:
  (a) a scheduler S, and
  (b) an algorithm A_j, which implements an Interface I_i.
  The distribution layer, D, is selected specifically to match the family of algorithms represented by A_j.

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

For a specific classic distributed algorithm the interface I_i is used by the scheduler to interact with all the different kinds of algorithms implementing that interface. Using an interface makes it easier to introduce new algorithms and to specify, at runtime, the algorithm that should be used.

For example, say the system has four nodes, and these nodes all need access to a shared resource — some for reading and some for writing. This obviously calls for the use of a distributed mutual exclusion algorithm. The classic algorithm, C, would be a ReaderWriter component. The Scheduler, S, would be the MEScheduler — the Mutual Exclusion Scheduler. The Interface I_i could be the MENT algorithm, for all Mutual Exclusion No Token algorithms. The algorithm A_j could possibly be the Maekawa or the Ricart-Agrawala algorithms — depending on the algorithm the system developer specifies should be used. The middleware could be Java RMI.

When the abstract architecture was implemented it became obvious that some auxilliary components were required in the system. These components

[1] Algon stands for *Algorithms On the Net*.

address two problems:

(i) The participating applications, with their algorithms, need to be identified uniquely to all other applications in order for them to be able to construct the request sets necessary for their correct functioning. Such identification supports middleware independence: some middleware frameworks, such as RMI, need to know the location of other objects, while others, such as CORBA, support dynamic discovery.

(ii) The Algon system aims to directly support monitoring of the system. The developer needs to be able to monitor system activity from a central point. It is necessary to provide the developer with a central display and control interface so that he/she can not only observe but also directly *control* activity in the system.

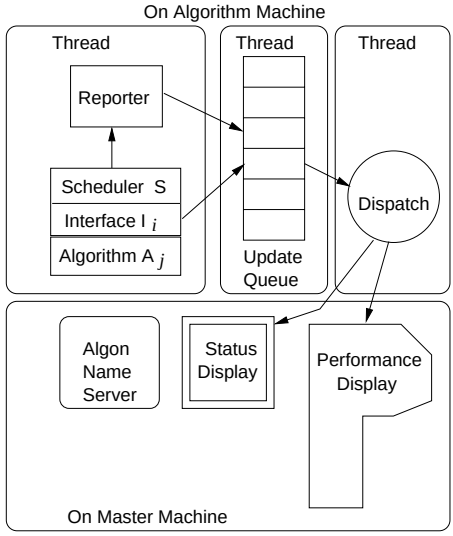

Fig. 2. *Minimising the Effect of Performance Measurement*

The first problem can be addressed either statically or dynamically. Assigning a name to each application statically is the simple solution, but this is not realistic in an industrial setting. Therefore we had to assign unique names dynamically. The obvious solution here is to use a name server, and the `AlgonNameServer`, shown in Figure 2, was created to serve this purpose. The application will instantiate the classic-algorithm-specific Scheduler, S, which will instantiate the algorithm, and then register itself with the `AlgonNameServer`. The name server will store the information about the participating application/algorithm and the IP address it is running on, and then assign a unique identifier to the application.

When the applications want to start using C, the `Scheduler` will ask the `AlgonNameServer` for the identifiers and IP addresses of the other participating applications and then start constructing request sets with those identifiers.

The second problem is addressed by having an `OutputDisplay` class. This is instantiated and bound by the `AlgonNameServer` so that all applications can send status information to one central output display, shown in Figure 3.

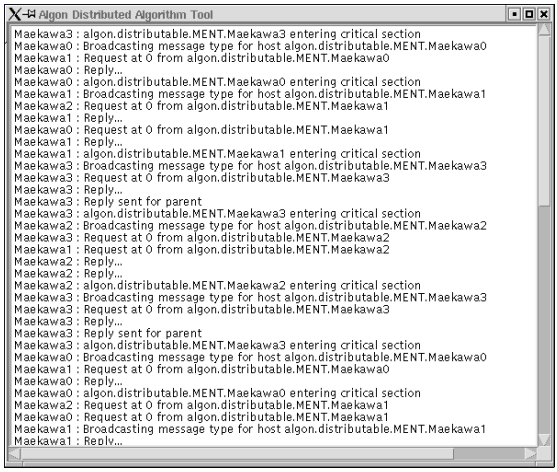

Fig. 3. *Output Display*

2.1 Performance Measurement

The output display mentioned in the previous section provides the distributed system developer with a coarsely-grained snapshot of system activity but it cannot provide any data to support realistic comparison between algorithms. The Algon em performance measurement component was created to this end. This component receives reports from all participating nodes so that it can construct meaningful graphs and tables to convey information about system performance to the programmer. In order to ensure that the measurement of system performance was done with minimal impact on the application a detached queueing system was used, as shown in Figure 2.

Algon reporting makes use of three classes — `Reporter`, `UpdateQueue` and `Dispatch`. The `Reporter` class is twinned with the `Scheduler` and the `Scheduler` reports on all interaction with the algorithm to the `Reporter`. The `Reporter` maintains all status information and constructs reports to be sent to the performance display tool. The reports are inserted into the `UpdateQueue`, which runs in a separate thread so that the `Reporter`, having placed the report

on the queue, can return control directly to the `Scheduler`. The `Dispatch` class watches the queue and when it detects a new report it removes it and sends the report to the performance display tool. The reports to the output display are also routed via the queue to minimise the negative impact of reporting on performance. In this way, we disassociate the time-consuming contact with the performance display tool from the `Scheduler`. The performance display is shown in Figure 4.

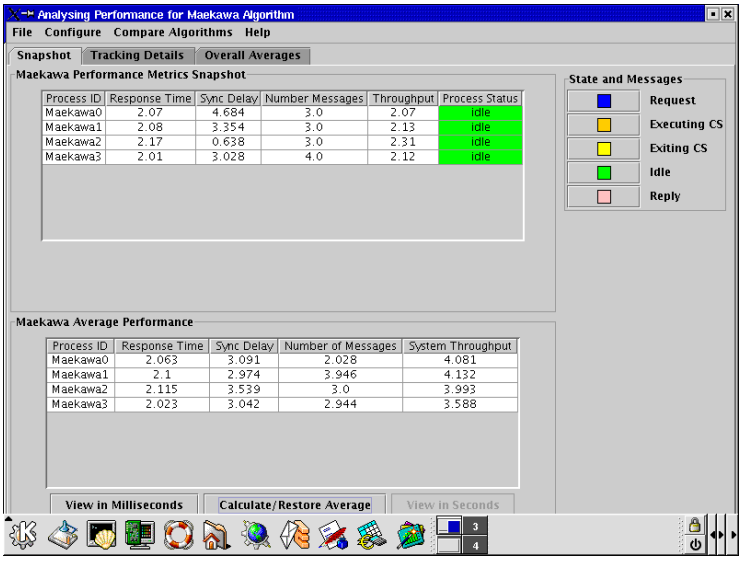

Fig. 4. *Performance Display Tool*

2.2 Dealing with Algon Failure

The failure semantics of distributed systems are different from centralised systems — hence the need for distributed algorithms. Distributed algorithms have been developed specifically to cope with such failures, and will report such failures in the Algon system by means of thrown Exceptions. However, it is important that the Algon layers do not introduce a whole new family of exceptions into the system caused by a failure of one of its components. The system could fail in the following ways:

(i) The Performance Display component: The system is protected from the failure of this component since the `Dispatch` class will not report any exceptions it deals with, but will simply stop sending reports to the tool. The functioning of the `Scheduler`, which reports to the tool via the `Reporter` and `UpdateQueue`, will not be affected by this failure. The

`Dispatch` class will continue to remove reports from the queue and then discard them so that the application will not be disrupted by the failure.

(ii) The failure of the `OutputDisplay`, while defeating the purpose of Algon, will not cause the application to fail. Once again the status reports will merely cease to be available.

(iii) The failure of the `AlgonNameServer` during system setup will cause the system to fail but failure after the initial setup will not affect the system in any way. We are aware that this single point of failure may be seen as a weakness in the system. It is, as is common in distributed systems, necessary to weigh up the advantages of having a dynamic registration mechanism with a single point of failure against a static inflexible failure-resistant naming system. We felt the former to be the better design choice. It is a relatively simple matter to replicate the `AlgonNameServer`, and this will be done if the need arises.

The whole Algon distributed layer and associated components have been developed with the philosophy that if an exception is caught the system will try to continue to function regardless so as to not interfere with the functioning of the application. Algon will therefore run with reduced Algon functionality in order not to sabotage the continued running of the controlling application. The following section shows how the Algon principles have been applied to incorporate distributed deadlock-detection algorithms.

3 Application of Algon

In previous work [3] details of a typical mutual exclusion algorithm's calling patterns were discussed. 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.

3.1 Distributed Deadlock Detection

Many algorithms exist in this category in the literature. According to Knapp [10], 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 5 illustrates how a distributed deadlock-detection algorithm is incorporated into a system with three nodes. Each node has a philosopher, *Ph*,

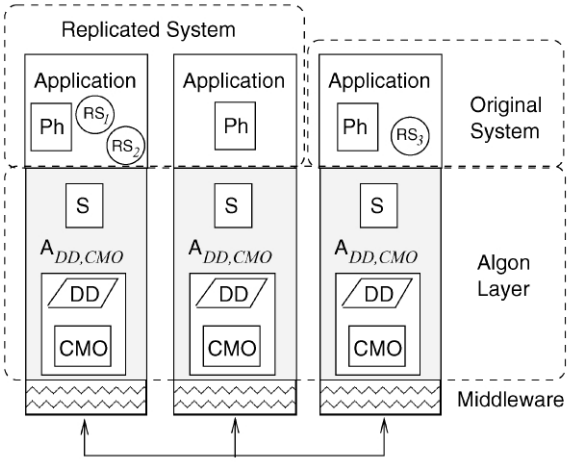

Fig. 5. *Using Algon to add a distributed deadlock detection algorithm*

and zero or more optional resources, RS_i. The Scheduler used for this algorithm is different from the Scheduler used for the Mutual Exclusion algorithms because of the specialised functionality that each Scheduler has to support. The Scheduler for deadlock detection algorithms is called the DDScheduler.

In order to use this algorithm it is necessary for the Algon framework to maintain a reference to each Philosopher from the DDScheduler inside the Algon framework. (The Mutual Exclusion algorithms do not need to implement a call-back in this way.) There is an apparent contradiction between our assertion of separation-of-concerns and this upward link from the `DDScheduler` to the `Philosopher`, as it appears to break the required separation. However, it must be borne in mind that any deadlock-detection algorithm needs access to the deadlocked processes and resources 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 just 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 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 relative frequency of deadlocks had been determined.

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);
  }
}
```

This code is recognisable as that which would appear in a centralised system, indicating the purpose of the component, C. For example, many transactions are required to request resources in a particular order to minimise the occurrence of deadlocks. This type of ordering is demonstrated in the above code fragment. However, even in with this kind of ordering deadlock can occasionally occur, especially when the resources are distributed and there is an inevitable time lapse between the application request for resources and the allocation of such resources to the application.

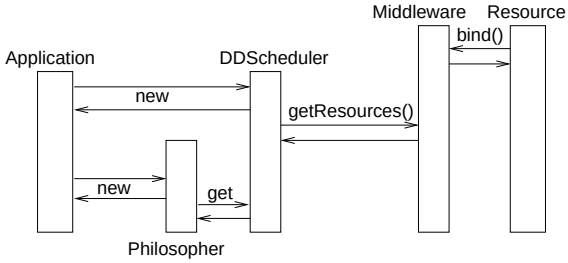

Fig. 6. *Setting Up with Deadlock Detection*

The interaction between the different participants in setting up the communication between participating nodes is shown in Figure 6. The resources are assumed to be available before the application starts executing. The setting-up process involves the application instantiating the `DDScheduler` and then instantiating the classic algorithm, in this case the `Philosopher`. All other details of the setting-up process are dealt with inside the `DDScheduler`.

Unlike other applications, such as the Readers-Writers problem which involves the distributed algorithm at each call [3], there is no need for the algorithm to be involved in the calls the `Philosopher` makes to `get` and `put` the `Resources`. The deadlock algorithm is only invoked when the current situation triggers an investigation into a possible deadlock. The interaction precipitated when a test for deadlock is triggered is shown in Figure 7. The

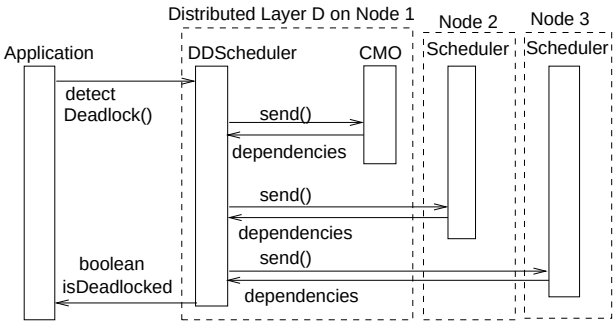

Fig. 7. *Testing for Deadlock*

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. Note that 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. We observed that the architecture designed for use in the mutual exclusion case had survived with minor changes when applied to a completely different genre of algorithm.

3.2 Other Classes of Algorithms

The algorithms and techniques that have been developed for use in distributed systems have been traditionally divided into five classes, namely mutual exclusion algorithms, deadlock detection algorithms, agreement protocols, resource management techniques and failure recovery techniques [26]. An algorithm is placed in one of these classes based on the effect which the algorithm achieves. Examining the implementation of the algorithms in a specific class reveals an interesting feature: they all expose the same methods. For example, the algorithms in the mutual exclusion class expose the methods enterCriticalSection, executeCriticalSection, and leaveCriticalSection. Deadlock detecting algorithms all expose the detectDeadlock method. By virtue of this feature, we can abstract away from the implementation details of all the algorithms in a class by having them all conform to one clean interface. This also simplifies the process of replacing the implementation that an application uses, *without changing the application itself.*

It is important to point out that, although a class of algorithms can be hidden transparently behind a clean interface, it is not possible to extend this

abstraction to cover all the categories. Put differently, no single interface can expose the functionality of, for example, both the mutual exclusion and deadlock detection classes of algorithms. The problem caused by this situation is that directly applying two (or more) interfaces in an application — where applying an interface includes adding whatever necessary state-maintaining logic to the application — would, for our purposes, result in too many changes to the client application. The different types Scheduler component (i.e. MEScheduler, DDScheduler, etc,) were introduced to ameliorate this problem. The schedulers are responsible for maintaining all state associated with the execution of an algorithm, and provide convenient hooks for the application to invoke the services of the algorithms. This has also allowed us to keep the architecture of the Algon framework consistent, regardless of the class (or classes) of distributed algorithm an application may choose to apply.

4 The Middleware Layer

In our initial implementation we used Java RMI as the middleware layer. Our intention was to concentrate on interchangeability of algorithms and provision of an algorithm performance visualisation tool before focusing on the comparison possibilities of the middleware layer that Algon could support.

It is desirable to provide a middleware-independent core of Algon classes because that makes it far simpler to extend Algon by incorporating another middleware implementation. This is important because it is useful to understand the impact the particular middleware is having on the system.

Making Algon middleware-independent is no easy task. To explain the difficulty consider the use of `java.rmi` as the middleware layer. If we wish to use the java tool `rmic` to facilitate remote invocation of objects representing distributed algorithms the stub class has to extend the java.rmi.server.UnicastRemoteObject class. This class provides the necessary server semantics required in order to support remote invocation. It also has to implement the `java.rmi.Remote` interface. All methods of any class implementing the `java.rmi.Remote` interface have to throw the `java.rmi.RemoteException`. This is an interesting and crucial design decision by Java's designers.

By giving remote invocation similar semantics to local invocation in Java the designers have attempted to provide a measure of distribution independence. Parameter passing in Java, happens in one of two ways: call-by-reference for object instances and call-by-value for primitive types. Remote invocation adds another way — call by deep copy — provided by object serialization. Since Java attempts to hide the remote nature of the invocation the programmer does not always know exactly how the parameters are being

passed, something he/she should know in order to refine code and make it more efficient.

An attempt to make the remoteness of the invocation transparent is bound to fail, however, because of the very real difference in the failure semantics of local and remote invocation. The programmer is therefore forced to make provision for possible failure of distributed components of his/her system, or the network linking him/her with that system, for remote invocations. Java forces the programmer to do this by ensuring that `java.rmi.RemoteException`s are thrown by all methods in a class which will be used remotely.

In order to explain the problems experienced in making the system middleware-independent we will expand upon the inheritance structure of Algon algorithms in Section 4.1, then discuss the difficulties in Section 4.2 and end off by introducing the `AlgonRmic` tool, which simplifies the process for the programmer, in Section 4.3.

4.1 Using the Mutual Exclusion Interface

The inheritance structure of the mutual exclusion algorithms used in Algon is shown in Figure 8. Each algorithm implements the particular interface which is implemented by all algorithms of a specific functionality type. For example, the `Ment` interface will be implemented by all **M***utual* **E***xclusion* **N***o* **T***oken* algorithms. The `Ment` interface is defined as follows:

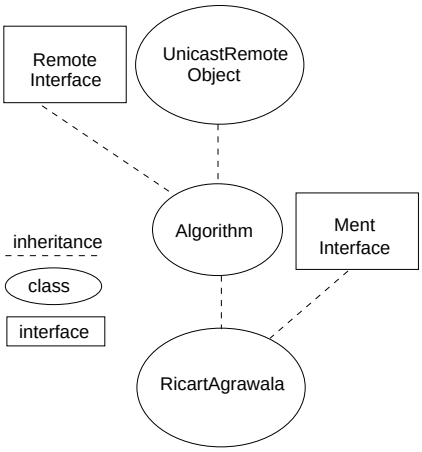

Fig. 8. *Algorithm Inheritance Structure*

```
public interface Ment extends Serializable {

  public void sendRequests(SchedulerInterface si) ;
  public void reply() ;
  public void request(long time,SchedulerInterface si, Ment m) ;
  public void getRequestSet() ;
  public void sendRelease() ;
```

```
  public void release() ;
}
```

In addition, in order to facilitate communication across the distributed system the specialisations of the `Algorithm` class also extends the `java.rmi.server.UnicastRemoteObject` and `Remote` classes. This ensures that the algorithms can have stubs generated for them and that the stubs can be distributed via the `java.rmi` middleware infrastructure. However, if we want to make the system middleware independent we can no longer have a `java.rmi`-specific implementation at such a high level in the system. We need rather to become middleware-specific at as low a level as possible and to generalise the upper inheritance structure.

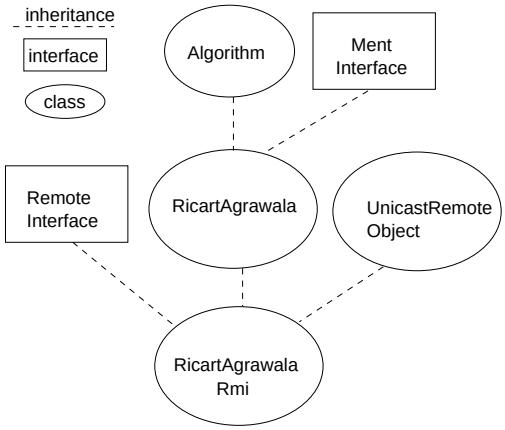

Fig. 9. *Extended Algorithm Inheritance Structure*

4.2 Making Ment Middleware Independent

Achieving independence is not as simple as it appears. It would appear that the solution would be to have middleware-specific implementations of each algorithm, which simply extends the inheritance structure as shown in Figure 9. However, this solution is flawed for two reasons:

(i) Multiple inheritance is not permitted in Java, therefore `RicartAgrawalaRmi` is not permitted to inherit from both `RicartAgrawala` *and* `UnicastRemoteObject`.

(ii) Java's exception handling mechanism has two provisos:
   (a) You *must* declare any exceptions that may be thrown and such a declaration *must* be part of a method signature.
   (b) A subclass which is overriding a method that throws an exception *must* declare that it throws that exception too — either the same

exception, or a subclass of that exception [16]. On the other hand, a method in a subclass cannot throw an exception unless its superclass method throws an exception of the same type. This feature has both advantages and disadvantages. On the one hand it is valuable to know what exceptions could be thrown by the methods of a Java class. On the other hand, the current restrictions prevent programmers from easily adapting systems in response to evolving needs.

`RicartAgrawalaRmi` must thus throw at least a `RemoteException` from all methods since it implements the `Remote` interface. Therefore the methods it overrides in `RicartAgrawala` must also throw a `RemoteException`, but this would defeat the desired middleware-independence of `RicartAgrawala`.

The only solution to the inheritance problem is to make use of delegation rather than inheritance. Thus a pairwise inheritance structure was established, where each algorithm has a middleware-dependent place-holder which is used as a stub class in the middleware and relays all algorithm-specific calls to the actual algorithm, as shown in Figure 10. The process can be automated as described in section 4.3

This deals with the multiple inheritance problem, but does not deal with the exception-handling problem, since both the middleware-*independent* algorithm `RicartAgrawala`, and the middleware-*dependent* algorithm, `RicartAgrawalaRmi`, must implement the `Ment` interface and since `RicartAgrawalaRmi` must throw `RemoteException` that means that `Ment` also has to throw `RemoteExceptions`, once again defeating the middleware independence of the algorithm.

There is a way to work around this problem. We cannot change the fact that the java RMI middleware-dependent algorithm throws a `RemoteException`. Therefore we have to ensure that the the `Ment` interface (which the algorithm implements) throws an exception of type `Exception`, which is a supertype of `RemoteException`. `Exception` is also a supertype of all other exceptions thrown in Java so that it will also act as a supertype for exceptions thrown by any Java-based middleware system. The implication of this is that the middleware-independent Algorithm has to catch and process exceptions for each of the Ment interface methods. This is not an unrealistic expectation, since the algorithms are essentially distributed components, and this fact will cause them to fail occasionally for different reasons. It is as well if the Algon system accommodates this reality of distributed systems.

To implement this delegation, Algon was extended by adding a `Middleware` interface which could be used by the system to invoke middleware-required functionality. This interface provides Algon with a uniform way of discover-

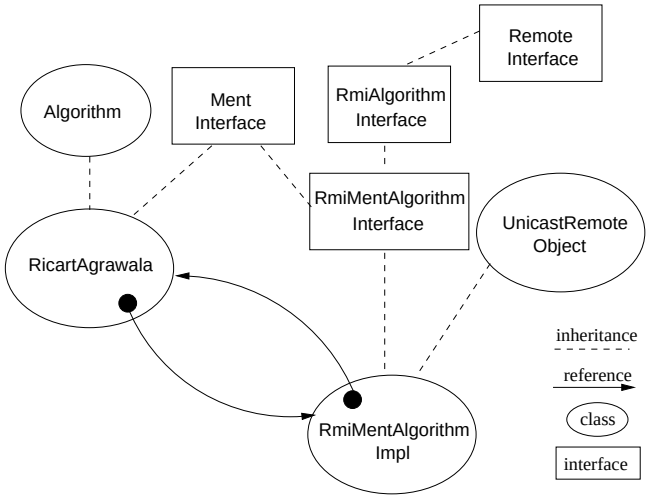

Fig. 10. *Pairwise Middleware-Independent Algorithm Structure*

ing, accessing and manipulating algorithm instances. This layer is illustrated in Figure 11. Algon detects the user's dynamic runtime middleware preference, instantiates the specific middleware implementation and provides a static reference which can be used by any class in Algon that needs to use middleware functionality. During instantiation all the middleware-specific components will be started up — such as the RmiRegistry in the case of `java.rmi`, for example. This frees the Algon user from the minute details required to ensure that all components that the specific middleware implementation requires are in place.

4.3 Automating the Process

The final step in making Algon middleware-independent was to define a new exception, called `MiddlewareException`, which could be used by all middleware layers to throw middleware-related exceptions.

Since the place holder classes (such as `RmiMentAlgorithmImpl`) are essentially middleware-dependent proxies for the actual algorithms, they are generated by a tool for `java.rmi` called `AlgonRmic`. For an interface such as `Ment`, `AlgonRmic` compiles the interface, and runs `rmic` on it so that the stubs and skeletons are ready for use by any algorithm implementing that interface.

5 Use of Algon

Two mechanisms are used to tailor runtime behaviour: configuration files and runtime variables. Configuration files are used to specify information that

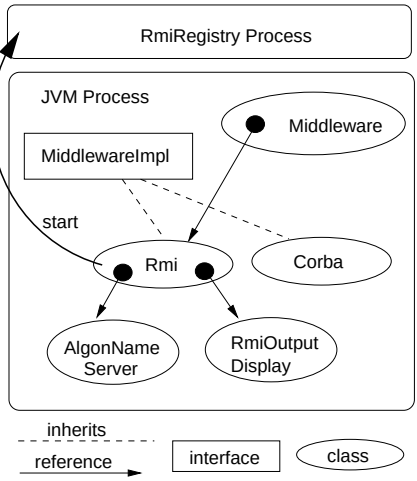

Fig. 11. *Algon Middleware Layer*

cannot be discovered from the runtime environment. The configuration file `hosts.prop` holds the following information:

- the IP address of the master site, where each application algorithm can expect to find the Algon Name Server;
- the number of algorithms that should be participating in the system; and
- the class name of the algorithm that the system should be using.

Runtime variables are used to specify behaviour that is tailorable per site. The following settings are currently available:

- generation of status output statements to the command line to signal critical changes in the system and assist debugging;
- choice of whether to measure performance or not;
- which middleware is to be used; and
- whether the Algon output should be sent to the command line or to the graphical user interface, as shown in Figure 3.

6 Related Work

Classifying Algon in order to draw comparisons with related work is not trivial. It has some similarities to reflective systems, and certainly applies separation of concerns techniques. It is also a very specialised programmer tool. A reflective system typically reasons about itself, and then acts upon itself based upon such reasoning [15]. This definition has been applied fairly loosely to

many different systems. Reflective systems are composed of a base level and a meta-level, with changes at the meta-level causing changes to the base-level's behaviour [30]. Algon cannot be classified as a reflective system since it does not react to its own behaviour, but rather to a runtime configuration setting. The configuration file cannot really be classified as a meta-object. Thus we cannot compare Algon to reflective systems. We therefore classify Algon as a *tool* which applies a *separation of concerns* technique to *algorithmic concerns*.

Some research has been done into providing programmers with tools which separate behavioural features of software from functional features [6,9]. The technique has been applied to a variety of different concerns, including real-time constraints [1], distribution and replication [6], exception handling [5], location control [17] and synchronisation [14]. There are basically three approaches to achieving separation of concerns:

(i) identifying the specification of concerns and allowing the programmer to specify each concern in a separate object. This can be done using proxies at compile-time [22] or by using reflection at runtime [30]. Another approach is the definition of aspects and the use of aspect-oriented programming [8,9]. Aspects are designed to address cross-cutting concerns across code. An aspect weaver then merges the aspect with the code to generate code where the concern is dealt with in a uniform way [21]. Zhang and Jacobsen have applied this to middleware architecture [31].

(ii) treating the concern as being orthogonal and freeing the programmer completely from it [20].

(iii) providing the programmer with a library which encapsulates the complexity [6]. The library typically contains functions which can be invoked by the programmer when required.

The first approach is usually done for reflective purposes but we do not feel that Algon is adding reflective capabilities to the system. Algon, however, has addressed the problem of middleware independence, and not a cross-cutting non-functional concern across a particular middleware system. Algon applies a modular approach, and does not make use of aspects. The second approach is also not suitable for Algon's purposes since the programmer must obviously be involved in the use of Algon. Algon is most similar to tools that fit into the third category. Algon does provide a library, but offers the programmer an additional level of choice, and supports an informed choice by means of the performance comparison tool.

One approach that fits into the third category and that also addresses distribution issues is Garf [6]. Garf provides the programmer with an extensible library for adding behavioural features to distributed programs. Whilst be-

ing innovative for its time, Garf has two shortcomings: it was implemented in Smalltalk which limits its applicability, and it does not attempt to offer a choice between different implementation techniques. Algon addresses distribution issues, as does Garf, but from an algorithmic perspective. Rather than providing a library of functions to be used blindly, Algon recognises the differing nature of distributed systems and offers programmers the capacity to tailor their systems accordingly.

7 Conclusions

In this paper we explain how the Algon framework operates, how the programmer can use it to interchange and compare algorithms, and how this concept can be extended so that the existing Java-RMI middleware can be dynamically replaced by CORBA. We showed that Algon does not require a knowledge of complex algorithms, and that the algorithms are interchanged with a minimum of effort.

In the development of the Algon framework, we have implemented three popular algorithms for achieving distributed mutual exclusion and for detecting deadlocks. These, however, form a very small subset of the algorithms proposed in the domain of distributed computing. Future work will include the implementation of a representative number of algorithms and techniques from all the aforementioned categories.

We have also outlined a technique for allowing interchangeable middleware. We have reason to believe the we will be able to extend the techniques outlined to other types of middleware as well. This work is ongoing. Thereafter we will extend the performance visualisation interface, currently tailored to reflecting algorithm performance, to assist in judging the performance impact of a specific middleware on an application. We also envisage a C# version of Algon in which CORBA can be interchanged with .NET or any other future middleware system. The lessons learnt through this research include practical aspects of software composition, the impact of language features on system design and issues associated with evaluating complex systems at runtime.

Acknowledgements

This work was funded by NRF grant 2053401 for Frameworks for Distributed Software. We also extend our thanks to the referees for the helpful comments which improved the paper.

References

[1] Aksit, M., J. Bosch, W. van der Sterren and L. Bergmans, *Real-time specification inheritance anomalies and real-time filters*, in: *[28]*, 1994, pp. 386–407.

[2] Ben-Ari, M., *Interactive execution of distributed algorithms*, ACM Journal of Educational Resources in Computing **1** (2001).

[3] Bishop, J. M., K. V. Renaud and B. Worrall, *Composition of distributed software with Algon — concepts and possibilities*, in: *Elsevier ENCS 65*, ETAPS Software Composition SC 2002, Grenoble, France, 2002.

[4] Chandy, K. M., J. Misra and L. M. Haas, *Distributed deadlock detection*, ACM Transactions on Computer Systems **1** (1983), pp. 144–156.

[5] Dellarocas, C., *Toward exception handling infrastructures for component-based systems*, in: *CBSE at SIGMETRICS*, ACM, Kyoto, Japan, 1998, pp. 141–150.

[6] Guerraoui, R., B. Garbinato and K. R. Mazouni, *Garf: a tool for programming reliable distributed Applications*, IEEE Concurrency **5** (1997), pp. 32–39.

[7] Kaveh, N. and W. Emmerich, *Deadlock detection in distributed object systems*, in: V. Gruhn, editor, *8th European ESEC and 9th ACM SIGSOFT FSE*, Vienna, Austria, 2001, pp. 44–51.

[8] Kiczales, G., *Aspect oriented programming*, ACM SIGPLAN Notices **32** (1997), pp. 162–162, no paper in volume, but table of contents includes an entry for this invited talk.

[9] Kiczales, G., E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W. Griswold, *An overview of AspectJ*, in: *ECOOP*, Budapest, Hungary, 2001, pp. 327–353.

[10] Knapp, E., *Deadlock detection in distributed databases*, ACM Computing Surveys **19** (1987), pp. 303–327.

[11] Kolehofe, B., M. Papatriantifilou and P. Tsigas, *Distributed algorithm visualisation for educational purposes*, in: *4th ITiCSE*, ACM, Cracow, Poland, 1999, pp. 103–106.

[12] Kramer, J., *Distributed software engineering*, in: *16th ICSE*, Sorrento, Italy, 1994, pp. 253–263, invited State of the Art Report.

[13] Lopes, C. and W. Hursch, *Separation of concerns*, Technical Report NU-655-95-03, Northeastern University (1995).
URL citeseer.nj.nec.com/lopes95separation.html

[14] Lu, J., M. Zhang, M. Xu and D. Yang, *A two-layered class approach for the reuse of sychnronization code*, Information and Software Technology **43** (2001), pp. 287–294.

[15] Maes, P., *Concepts and experiments in computational reflection*, in: *OOPSLA*, ACM, Orlando, Florida, 1987, pp. 147–155.

[16] Mikhailove, A. and A. Romanovsky, *Supporting evolution of interface exceptions*, in: A. Romanovsky, C. Dony, J. L. Knudsen and A. Tripathi, editors, *Advances in Exception Handling Techniques.*, number 2022 in Lecture Notes in Computer Science, Springer-Verlag, 2001, pp. 94–110.

[17] Okamura, H. and Y. Ishikawa, *Object location control using meta-level programming*, in: *[28]*, 1994, pp. 299–319.

[18] Ossher, H. and P. Tarr, *Using multidimensional separation of concerns to (re)shape evolving software*, CACM **44** (2001), pp. 43–50.

[19] Parnas, D. L., *On the criteria to be used in decomposing systems into modules*, CACM **15** (1972), pp. 1053–1058.

[20] Prentezis, T., "Management of long-running high-performance persistent object stores," Ph.D. thesis, Department of Computing Science, Department of Computing Science, University of Glasgow (2000).

[21] Pulvermüller, E., H. Klaeren and A. Speck, *Aspects in distributed environments*, in: *First International Symposium on Generative and Component-Based Software Engineering. GCSE'99*, Erfurt, Germany, 1999.

[22] Renaud, K., *Experience with statically generated proxies for facilitating java runtime specialisation*, IEE Proceedings - Software **149** (2001), pp. 169–178.

[23] Renaud, K., J. Lo, J. Bishop, P. van Zyl and B. Worrall, *Algon: a framework for supporting comparison of distributed algorithm performance*, in: *11th Euromicro PNDP03*, Genoa, Italy, 2003, pp. 425–432.

[24] Ricart, G. and A. K. Agrawala, *An optimal algorithm for mutual exclusion algorithms*, CACM **24** (1981), pp. 9–17.

[25] Schreiner, W., *A java toolkit for teaching distributed algorithms*, in: *ITiCSE*, ACM, Aarhus, Denmark, 2002, pp. 111–115.

[26] Singhal, M. and N. G. Shivaratri, "Advanced Concepts in Operating Systems," McGraw Hill, 1994.

[27] Tel, G., "Introduction to Distributed Algorithms," Cambridge University Press, 2000, 2nd edition.

[28] Tokoro, M. and R. Pareschi, editors, "Object-Oriented Programming, Proceedings of the 8th European Conference, ECOOP '94, Bologna, Italy," Lecture Notes in Computer Science **821**, Springer, 1994.

[29] Trinder, P., K. Hammond, H.-W. Loidl and S. L. Peyton-Jones, *Algorithm + Strategy = Parallelism*, Journal of Functional Programming **8** (1998), pp. 23–60.

[30] Welch, I. and R. J. Stroud, *Kava — using byte code rewriting to add behavioural reflection to Java*, in: *COOTS '01*, San Antonio, Texas, USA, 2001, pp. 119–130.

[31] Zhang, C. and H. Jacobsen, *Quantifying aspects of middleware platforms*, in: *Proceedings of the 2nd International Conference on Aspect-oriented software development* (2003), pp. 130–139.