A Flexible Framework for Consistency Management in Object Oriented Distributed Shared Memory

S Weber and P A Nixon and B Tangney
Distributed Systems Group,
Department of Computer Science,
Trinity College,
Dublin 2,
Ireland.
{sweber,pnixon,tangney}@cs.tcd.ie

October 13, 1998

Recent Distributed Shared Memory (DSM) systems provide increasingly more support for the sharing of objects rather than portions of memory. However, like earlier DSM systems these distributed shared object systems (DSO) still force developers to use a single protocol, or a small set of given protocols, for the sharing of application objects. This limitation prevents the applications from optimising their communication behaviour and results in unnecessary overhead.

A current general trend in software systems development is towards customisable systems, for example frameworks, reflection, and aspect oriented programming all aim to give the developer greater flexibility and control over the functionality and performance of their code. This paper describes a novel object oriented DSM framework that offers the developer the ability to customise the DSM **consistency model** and the underlying DSM **coherency protocol** on a per object-basis. The framework allows the developer to exploit application specific semantics, and thus improve performance, by enabling different consistency models and coherence protocols to be used within a single application. The framework is implemented in JAVA and performance of a number of applications that use the framework is reported.

1 Introduction

Recent advances in Distributed Shared Memory (DSM) systems are providing increasingly more support for the sharing of objects rather than portions of memory, e.g. Bal *et al* [2] and Guesdes and Castro [14]. While the distributed shared object (DSO) approach allows a finer control over the granularity of sharing than traditional page based systems, as in Amza *et al* [1], it still forces the developer to use a single protocol, or a small set of given protocols, for the sharing of application objects. This limitation unnecessarily constrains the application developer, preventing them from optimizing the communication behaviour of their application thus incurring avoidable overhead.

Apart from the granularity of sharing, the other main way in which DSM systems differ from each other is in the way in which they control access to, and updates of, shared data. The majority of DSM systems provide the developer with a single protocol. For example with **sequential consistency**, as defined by Lamport [18], every access to shared data is ordered individually. This ordering removes the necessity from the developer to declare explicitly the start and end of blocks of shared data accesses and provides the developer with transparency. An alternative protocol is Midway's **entry consistency**, [5], which offers the developer enhanced performance by bundling a series of shared data accesses to a block. The bundling of accesses reduces the number of update messages and thus the network traffic.

However, as pointed out by the developers of Munin [4], not all data items in all applications lend themselves to a single protocol. Munin defines nine characteristic ways in which data is shared and provides support for each type of sharing.

This paper describes an approach to providing support for DSM in which the granularity of sharing is at the object level and the protocols used to maintain consistency are customisable by the application developer. In particular it describes an object oriented framework that allows application developers to customise the underlying DSO system to suit the specific needs of their applications.

Other projects [16, 10, 11] have aimed at providing flexibility by allowing customisability of consistency policies at the user level but until now no project has fulfilled our vision of a complete customisable system where the developer may change all aspects of the shared memory from consistency policies and memory management to communication issues. Moreover no other project provides this flexibility on a per object basis, allowing mixed consistency models and coherence protocols to be used within a single application.

The layout of the body of the paper is as follows: section §2 discusses how DSM systems can be described in terms of a consistency model and a coherency protocol; section §3 describes the need for flexibility in DSM systems and the approaches that other researchers have taken to providing it; sections §4 and §5 then go onto to discuss our approach to flexibility by describing the the design and implementation of our DSO framework; a discussion of the performance of some applications that use the framework is given in §6 while the conclusions that that be drawn from our work are given in §7.

2 Consistency Maintenance

The approach taken, in this paper, to providing a flexible DSM system is to characterize the functionality of any DSM system as having two distinct components - a consistency model and a coherency protocol. The **consistency model** defines the level of consistency provided by the system while the underlying **coherency protocol** is responsible for managing data so that the required level of consistency is actually provided. The relationship between the two is shown in figure 1. This approach is novel but we believe it to be general enough to encompass the approaches taken by most DSM systems to date and it is this novel approach that is the key insight at the heart of the framework described in §4.

The following two sections elaborate upon what is meant by consistency model and coherency protocol, while §3 discusses the related work in the area in terms of our characterisation.

2.1 Consistency Models

Consistency models define the order in which accesses to shared data are seen by interested parties and a number of different models have been proposed in the literature such as *sequential consistency*, *entry consistency* and *lazy release consistency* [1]. This paper argues that, in general, all consistency models can be characterised by four properties: order of access; concurrency; atomicity and scope.

- The **order of access** defines the sequence in which accesses are seen by interested parties. This property represents the heart of the consistency model. In general, the order can be seen from two points of view. The order in which the originating node sees its own operations and the order in which other nodes see operations.

  For example sequential consistency and processor consistency both define that all accesses have to be seen in the order they are stated in the program. Sequential consistency goes further to define that every node has to see every access from all nodes in exactly the same order. Processor consistency on the other hand defines that accesses issued by one node have to be seen in the same order from every node. However, accesses by two nodes can be seen by different nodes in different orders.

- **Concurrency** of access defines if nodes can concurrently access data and the modes in which they can access it. Read operations can usually be executed concurrently without causing problems while under certain circumstances writes may be executed concurrently - e.g. if the writes are to non-overlapping data. In all there are four possible combinations for concurrent read write access, namely: EREW (exclusive read, exclusive write); CREW (concurrent read, exclusive write); ERCW (exclusive read, concurrent write) and CRCW (concurrent read, concurrent write).

- **Scope** determines the set of data that is to be kept consistent. Some algorithms assume the order valid for all shared data while other algorithms restrict the order to a certain set of data [5]. For example sequential consistency commonly assumes the order of accesses valid for all shared data. Entry consistency on the other hand defines the order of accesses only for the individual data item.

- **Atomicity** defines whether the propagation of updates is done on a per access basis or whether several local updates can be done before a batched update is sent out. For example sequential consistency orders all accesses on a per access basis. In release consistent systems it is possible to combine accesses in a block.

These four properties are sufficient to describe the implementation of any consistency model. The properties determine the nature of the results that are to be expected from the consistency model and the connection of the consistency model to the underlying coherency protocol component.

2.1.1 Examples

Two derivations of lazy release consistency have been described by Keleher [16] and Amza *et al* [1]. The first version uses a barrier mechanism and the second a locking mechanism, as the synchronisation mechanism. The consistency models can be described with the properties as follows. For both models the atomicity is a block of accesses and the scope of consistency embraces all shared data in a system. The difference between the two models is captured by the order of accesses. The model using a lock mechanism forces the block accesses to happen sequentially. The model using a barrier mechanism forces no ordering among the access blocks. This relaxation is possible if no data races are expected in applications. It enables the system to implement different kinds of concurrency as described by Keleher [16].

An implementation of sequential consistency as described by Mizuno *et al* [20] can be characterised as a system where all accesses are ordered sequentially. The scope of the model embraces all data shared by the system. The concurrency of accesses is defined as CREW and the atomicity is defined on a per access basis.

2.2 Coherency Protocols

Coherency protocols define the management of shared data. This includes satisfying both access and synchronisation events. An **access event** is either a read or a write to a local copy on a node while a **synchronisation event** is a request to synchronise all copies of a data item.

To handle these events, a coherency protocol can be be subdivided into three components: memory management; ownership and synchronisation as shown in figure 2.

- The **Memory Management** component controls access to local copies of shared data, access to which can be divided into access events and update incorporation. Access events are either read or write operations that are issued by local applications. Update incorporation describes the process of incorporating update information from other local or remote copies.

- The **ownership** component is responsible for liasing with other nodes to keep track of the current "ownership" of the data. A variety of algorithms have been proposed in the literature, e.g. *static central ownership management* [19] and *directory-based ownership management* [7].

- The **synchronisation** component is responsible for the management of synchronisation events and it in turn generates either update propagations or an update requests. An update propagation is sent to remote nodes in order to inform them about changes to shared data. It contains a revision of the shared data that is retrieved from the memory management component. Along with revision data, an update propagation holds protocol-related data, like time stamps or address information.

  On receiving an update propagation from a remote node the synchronisation component has to determine if the revision is relevant to its local copy and, if necessary, to ask the memory management component to incorporate the revision of the data.

  An update request is a demand to synchronisation components on remote nodes to propagate modifications to their local copies. It may contain a set of protocol-related information, and is sent to nodes that are determined by the ownership component. The request is answered by an update propagation.

3 Related Work

Within the computer science community at large there is a strong push towards providing flexible systems. At the application level there is, for example, the recent trend towards component based programming while within the operating systems arena a variety of techniques aimed at providing flexibility are being investigated. These range from the use of frameworks, as exemplified in Choices [8], to the use of reflection as pioneered in Apertos [23].

The flexibility in most systems is achieved by opening up the system's internal structures and making them available to the developer. The two steps of this process have been subject of prior research in the DSM community and the most pertinent systems, from the point of view of this work, are discussed in §3.1. Before that, however, there is a brief discussion of how some other authors in the field have used the terms consistency model and coherency protocol. This is included as there is some overloading of the terminology in the area.

- Shi et al investigate in [21] the interaction between consistency models and coherency protocols. The conclusion of their investigation is that a consistency model is a combination of an event ordering mechanism and a coherency protocol. This conclusion contradicts our approach which holds that a consistency maintenance algorithm is a combination of implementations of consistency models and coherency protocols. However, closer examination shows that the difference between the two forms is only a difference in the definition of the individual terms. The term "consistency model" used in [21] matches the term consistency maintenance protocol used here and the term "event ordering" matches the term "consistency models" used in this paper.

- Gharachorloo et al in [13] give formal specifications for a set of consistency models. The specifications define in detail the conditions of the consistency models. The conditions however are only proof that an implementation ensures a defined level of consistency. They do not fully describe the implementation of a consistency model to the developer.

3.1 Flexibility

To date a number of approaches have been attempted for providing flexibility in the DSM area, the most notable of which are outlined below.

- **Munin** [3, 9] offers the application developer a set of consistency maintenance algorithms. Application developers can chose from the provided set of algorithms on a per data item basis.

  The algorithms are a close-coupled combination of consistency models and coherency protocols. Each algorithm is targeted to a group of shared objects with specific characteristics. The algorithms include provision for types as result objects, migratory objects, producer consumer objects, etc.

  Munin differs from our framework in that it only provides a limited set of algorithms. These algorithms are fixed in the system and can not be adjusted to application needs, as is made possible with our framework. All types provided by Munin however can be implemented using in our framework.

- **CVM** [17] implements a shared memory systems based on a user-level library. The library provides the developer with primitives for synchronisation and allocation of shared memory. Memory that is allocated with the provided primitives is consistent at synchronisation points that use the given synchronisation primitives. The user can choose, with command-line arguments at program start, between a set of given consistency models.

  The primitives are based on abstractions for memory pages and coherency protocols. The implementation of the library can thus modified to work with different protocols.

  CVM comes very close to our vision of a complete customisable system by opening the components of the system to the developer. It lacks however the possibility of using a combination of different consistency models for different types of shared data. We see this as a major advantage in our system and a necessity in order to be able to provide fine grained control over the sharing characteristics.

- **Disom** [10] presents an object-oriented framework for object sharing. Classes are made sharable by inheriting from particular super-classes. These super-classes define methods that facilitate the exchange of updates among interested nodes. On top of these methods different consistency models may be implemented.

  The framework provides the developer with an implementation of entry consistency. Additional consistency models can be added. The details of communication can be modified by the developer and adjusted to the individual application.

  Disom differs from our framework in two major ways: the limitation of sharing types to classes and the restriction to a single communication protocol. We see it as essential to be able to share instances of the same class with different sharing characteristics. Further the developer should be able to change to a suitable communication mechanism whenever necessary.

- **Rabbit** [15] is similar to DISOM in that it provides an object-oriented framework for object sharing. The framework provides an interface for consistency maintenance of the framework and the developer can -

based on this interface - define application specific consistency maintenance algorithms. The methods defined by the interface are called by the underlying system whenever a sharing event occurs. This mechanism allows the developer to implement sharing characteristics adjusted to the individual class.

The difference between Rabbit and our framework is, as with Disom, the limitation of sharing characteristics to the class level and the limitation of the underlying coherency protocol.

- **Arias** [11] provides the developer with a single address space system based upon a 64-bit virtual address. The system itself does not enforce consistency among multiple copies of shared data but provides primitives which support synchronisation. Various consistency models can be implemented using these primitives.

  Being a single address space system confines Arias to a single distribution management algorithm for the coherency protocol and thus does not allow the communication to be adjusted to reflect certain sharing characteristics.

4 Consistency and Coherency Framework

A framework consists of a collection of classes together with a number of patterns of collaboration among instances of these classes[6]. The main components of our framework are the consistency model and coherency protocol. These components are supported by components for communication and concurrency control. A high level view of the main components is given in figure 3 with the full class hierarchy for the consistency model being given in figure 4.

4.1 The Consistency Model Framework

The detailed framework presented in figure 4 follows on directly from the characterisation of consistency management given earlier in the paper.

All consistency model components are derived from a single base class. This base class defines the common interface for all consistency models and supports four fundamental operations to access shared data: read; write;

lock and unlock. The access operations are mapped to corresponding operations provided by an underlying coherency protocol. The synchronisation operations are mapped to corresponding operations provided by concurrency control.

Consistency models can be sub-divided into two major categories: strict and relaxed models. Strict consistency models order each access operation individually while relaxed models combine a set of access operations and impose an order on these sets.

The two sub-categories are represented in the inheritance hierarchy by two classes. The class for strict consistency models redefines the access operations that are inherited from the base class. The access operations are modified so that they interact with a given ordering mechanism, e.g. locking mechanism, if necessary. The class for relaxed consistency models defines two new operations. These two operations are used for annotation in the program to signify the beginning and the end of a set of access operations.

The next level in the inheritance hierarchy consists of ready-to-use implementations of various consistency models. These classes implement the specific characteristics of the individual consistency model. The developer may of course add their own models.

The class for sequential consistency requires that one of the components connected to it supplies a global sequencer. This sequencer has to ensure that all access operations are executed sequentially in a global context. The sequencer represents just the functionality. It is not defined how this functionality is achieved. The sequencer can be represented for example by a concurrency control component which locks the access to resources in a global context. Another possibility would be a coherency protocol component which has a central server that orders all accesses.

4.2 Coherency Protocol Framework

As shown in figure 5 the coherency protocol component is built from subcomponents for memory management, ownership management and distribution management. The methods defined by the general interface are used as wrappers around the 3 components to provide the individual behaviour of a particular coherency protocol. Two structures are exchanged by these components and contain the fundamental information that is passed between nodes. These and other two structures are described in following section.

4.2.1 Data Structures

Descriptor

A descriptor holds references to locations in the local copy of shared data. References may describe one location like a single field or memory address or a set of locations or ranges of an array. A location signifies the smallest sharable unit, e.g. a unique memory address or a field.

The implementation of references depends on the implementation languages and environment. An implementation in C/C++, for example, may favour references in the form of low level addresses whereas the implementation in Java, described in §5, favours field identifiers and array indices.

Update

An update is a collection of descriptors and data. An update holds descriptors that point to various locations in local copies of shared data. In these locations the corresponding data from the update have to be inserted in order to update a local copy.

The internal structure of an update depends on the following characteristics:

- An update may contain either a complete copy of a shared data item or hold a set of values that have been changed since the last synchronisation event.
- The internal structure of data depends on the implementation language and the level at which memory can be accessed. The data may for example consist of memory pages that are platform specific or of some form of encrypted data that could be platform independent.
- An update also depends on the protocols implemented by the distribution management component. In the case of an invalidation algorithm it may hold no data at all, in which case the descriptors of this message point to the locations that have to be invalidated.

Intention

An intention represents a combination of an identifier and a descriptor. The identifier specifies the operation that is to be executed and the descriptor, as given previously, describes an area of shared data that are effected by the operation. Based on this information intentions can determine if they conflict with any other given intention.

Intentions are usually defined by reference to a class of shared data items and use information obtained from these items. They are used in all components of the framework to determine what actions have to be taken when an intention is announced or when an intention finishes.

4.2.2 Types of Coherency Protocol

There are two flavours of coherency protocol. Update, or push, protocols send out updated data while invalidate, or pull, protocols mark the data as invalid requiring it to be refreshed on a subsequent access. Coherency protocols also differ in the way they distribute data which can be done either by sending a complete copy of the data or by sending the incremental changes only. Our framework for coherency protocols is shown in figure 5.

The coherency protocols implement a general interface that allows the consistency model to communicate with different coherency protocols. The core methods of this interface are *forceUpdate* and *ensureConsistency*. The method *forceUpdate* distributes updates. First it determines what needs to be updated by asking the memory management component. Then it asks the ownership management component for the destination of updates. The update is sent to its destination by using the sendUpdate method of the distribution management component. The *ensureConsistency* method asks the ownership management component to determine the source for updates. This source is then asked for an update using the *requestUpdate* method of the distribution management component. The returned update is then incorporated into the local copy by the memory management component.

4.2.3 Memory Management

The memory management component is responsible for the management of the local copy of a data item and it carries out two major functions: the processing of access operations and the handling of updates.

The processing of access operations comprises read and write operations to the local copy, all synchronisation is to be done by the connected consistency model.

Write operations cause the memory management component to update a location of the local copy with a given value. The memory management has to resolve a given descriptor into a location and write the given value into this location. This definition provides support for single values as well as for a combination of objects likes arrays or object compounds. The write operation can be used to keep a list of changes for a local copy. This list can contain only the locations changed in the local copy or a list of all changes and values changed.(A complete list, rather than one containing deltas or diffs, may provide information for a roll-back in fault-tolerant systems.)

The memory management component provides methods to determine and incorporate updates from respectively in a local copy of shared data. These methods - *determineUpdate* and *incorporateUpdate* - use descriptors that are given as parameters to extract from and incorporate data into a local copy. The *determineUpdate* method asks a given descriptor to extract the data it describes from a local copy and gives an update containing the extract and the descriptor pair back as result. The *incorporateUpdate* method takes a descriptor from a given update and asks the descriptor to restore the data connected to it into the local copy.

4.2.4 Distribution management

The distribution management component is responsible for the exchange of data between copies. It is linked closely to the communications mechanism. The distribution management component provides methods to send and request updates from remote nodes. The method to send updates - *sendUpdate* - takes as parameters an update and a set of destinations. The updates are sent to the addresses of the given destinations. The method to request updates - *requestUpdate* - takes as parameters a descriptor and a set of sources. The sources are contacted in turn to retrieve updates from them according to the given descriptor. Every distribution management component, when started, listens to a given address for incoming requests and updates.

4.2.5 Ownership management

The ownership management component is responsible for the administration of the ownership of copies. The main activity of the ownership management is to identify the location of other copies of shared data.

The interface to the ownership management component provides three methods to enable this service: a method to announce that a local copy is being shared (*registerCharge*), a method to determine the source of updates (*getUpdateDestinations*), and a method to determine the destination of updates (*getUpdateSource*). The exact definition of these methods depends on the distribution management and the underlying communication mechanism.

5 Current Implementation

The current implementation consists of a set of abstract Java classes that implement a number of different consistency models and coherency protocols.

5.1 Consistency Models

The implementation of the framework for consistency models is made up of generic super-classes at the base level and a small set of consistency models. The set of consistency models contains **Lazy Release Consistency** and a number of variations on **Weak Consistency**.

The generic super-classes provide the connection between the actual implementation of a consistency model and supporting components. The supporting components are represented by underlying coherency protocol and concurrency control components. The base class for consistency models stores references to these components for use by the derived classes.

A superclass for relaxed consistency models is derived from the base class explained above. This class defines the interface for all relaxed consistency models. The interface consists of two methods to mark the beginning and end of a critical section; *acquire* and *release*. The *acquire* method takes as an argument an *intention* containing a description of the region of shared data which will be accessed during the critical section. The *release* method indicates the end of the *intention*. Both methods have to be defined in derived classes to model the individual behaviour of a relaxed consistency model.

5.1.1 Weak Consistency

Two minor derivations of weak consistency have been implemented, both are based on the class for relaxed consistency models.

WC1 is based on a concurrency control component which provides a mutex lock mechanism. The *acquire* method announces the beginning of a critical section by acquiring a lock from the concurrency control component. The intention given as parameter to the method is used in the acquiring process as the parameter for this action. This way the concurrency control mechanism can determine if the intended action conflicts with another ongoing action. The *release* method ends the critical section by notifying the coherency protocol to publish changes made to the data. After the changes have been published the lock is released.

WC2 is based on a barrier mechanism. The implementation assumes that changes to the shared data do not affect each other; or if they do that they are synchronised by the developer. The *acquire* method stores the given intention for later use in the *release* method. The *release* method publishes the data described by the descriptor of the intention. After the data has been published, the method waits at a barrier until all participating parties have published their changes to the data.

5.1.2 Modified Weak Consistency (MWC)

A modified version of weak consistency has been implemented to facilitate the sharing characteristics of the Life application, explained later, which requires the use of two coherency protocols. This requirement can be met in two ways. By using two consistency models where each consistency model is connected to a different coherency protocol or by implementing a consistency model that is based on two coherency protocols.

In order to keep the implementation of applications the same regardless of what consistency model is used we opted to implement a modified weak consistency model. This consistency model takes as parameters two coherency protocols. The first coherency protocol exchanges data between a central copy and others. The second coherency protocol exchanges data between all copies of shared data.

The location of the data exchanged in this application does not change with every iteration. Thus the descriptors in this implementation have been embeded into the consistency model and are initialised at startup time. The *acquire* method has been redefined not to change these descriptors. The *release* method has been defined to invoke the coherency protocols to publish the updates. Each coherency protocol is invoked with a corresponding descriptor.

5.1.3 Lazy Release Consistency (LRC)

The implementation of lazy release consistency is similar to the weak consistency implementation in the relaxed consistency base class. To model the behaviour of lazy release consistency the class redefines the *acquire* method to ask the coherency protocol to ensure that the local copy is up to date. The *release* method tells the coherency protocol which data items have been changed during the critical section.

5.2 Coherency Protocols

The coherency protocol is made up of the memory management, distribution, and ownership components. The later two interface closely with the communication service. A number of different coherency protocols have been implemented, all using the same memory management component. They differ primarily in the communication service they use which in turn results in different distribution and ownership components.

The memory management component provides full copies for updates[1]. This full copy can be taken from the entire shared data item or from parts of it. Other memory management components could implement a mechanism to provide diffs or updates representing an invalidation of other local copies. To provide diffs a component would need to compare copies made before changes occurred to a local copy and this would need to be done when an update is to be determined. To provide an invalidation protocol the mechanism needs only to return the descriptors of the changes that have been made.

The details of the four coherency protocols that have been implemented are as follows: the first two are based upon a centralised scheme while the last two are based upon keeping peer replicas up to date. All protocols define wrapper methods called *forceUpdate* and *ensureConsistency* to represent their behaviour.

The code dependent upon the individual communication mechanism has been encapsulated in the distribution management and the ownership management components.

CP1 uses a TCP socket as communication mechanism between the central and other copies. The ownership management registers the incoming

[1] *Full copies* refers in this context to a copy of the changed data rather than a difference,or `diff`, as defined by Munin or Treadmarks.

address with a registry. Ownership managers of other copies retrieve the location of the master copy from this registry and provide the distribution management with this address as source and destination for updates.

CP2 uses a UDP socket implementation provided by the *iBus*[2] package as a communication mechanism between the central master copy and the others. The ownership management of this protocol is similar to the one used in CP1. The major difference is the change from the handling of socket addresses to the handling of iBus URLs.

CP3 uses multicast over UDP to keep replicated copies up to date.

CP4 uses multicast over iBus to keep replicated copies up to date.

6 Performance

This section reports on the performance of two parallel applications, matrix multiplication and Conway's game of life, developed using the framework. For each application, a number of versions were implemented using different combinations of consistency models and coherency protocols.

Neither application is trivially parallel. Both require significant communication between the workers. All tests were performed on a number of SUN workstations[3] connected by a 10Mbps Ethernet. The timings presented are averaged over at least 20 executions and all times were taken at off-peak periods when both the network and the workstations were lightly loaded.

6.1 Matrix Multiplication

Matrix multiplication, of the form $A = B \times C$, was parallelised using the "give me work" paradigm. The master creates the 3 arrays. Workers and the master share full copies of A and C while B is only partially shared, in that workers only need access to selected rows. Slaves request work items from the master and on completion of a computation the workers write the result into the destination matrix and then contact the master looking for

[2]iBus provides a group communication mechanism for Java applications. It supports UDP multicast, UDP point-to-point and TCP point-to-point communication[22].

[3]Model - Sparc Ultra-1 166MHz, 64MB memory.

more work. Each computation involves getting a full row from matrix B and multiplying it by all the columns in C to produce a full row in the result matrix A. Synchronisation is via mutex locks or barriers which were provided as part of the synchronisation component.

Four different versions of the application were written. All versions use WC2 and CP3 for sharing matrix C and LRC and CP2 for matrix B. The difference between the four applications is in the sharing characteristic defined for the result matrix (A). In the first version of the application A is shared with WC1 and CP4, in the second with WC1 and CP2, in the third with WC1 and CP3, and in the fourth with LRC and CP1.

The speed-up curve obtained for multiplying two 600x600 element arrays is given in figure 6. The first version shows less speed-up because of the significant overhead incurred for group management and reliability in CP4. Versions 2, 3 and 4 show improved performance with version 4 being marginally better than the other two.

6.2 The Game of Life

Conway's "Game Of Life", [12], involves populating a two dimensional grid with a pattern of full and empty cells. Successive iterations of the grid are produced in which the cells live or die in the next generation depending upon the number of immediate neighbours they have. Life can be parallelised by dividing the grid into a number of strips and processing each strip in parallel. Access to adjacent strips is required when calculating the values of cells on the edges. The state of the grid is displayed by a master. In each iteration the master receives a full copy of the data but as it is too time consuming to display every iteration, it takes longer to display the data than it does to calculate it, only every 10th iteration is displayed.

From the point of view of this paper Life has the interesting property that the optimum coherency protocol for sharing data between workers is different than the one for sharing data between workers and the master. Hence three different versions of Life were implemented:

- Life1 used the weak consistency model WC2 combined with the multicast coherency protocol CP4 for the sharing between all nodes, i.e. for both worker-worker sharing and worker-master sharing.
- Life2 used the modified weak consistency model MWC combined with two coherence protocols CP1 and CP4. CP1 was used for sharing

between worker and master while CP4 was used to distribute the edges of strips between workers.

- Life3 is similar to Life2 but used CP3 instead of CP4. This change led to significant better performance because of less overhead for group administration and reliability.

The resulting speed-up curves are shown in figure 7.

7 Conclusions

This paper presents a highly flexible Java framework for distributed shared objects based on a per object sharing mechanism. The crux of the framework relies on our division of sharing policies into consistency models and coherence protocols, and the subsequent general characterisation of these two elements. By adopting this approach we are able to provide a framework that allows users to extend the set of available protocols by implementing their own. Furthermore not only is it possible to have different coherency and consistency protocols within a single application but it also possible to have the sharing of a single object controlled by different protocols as demonstrated in the Life example. This represents a significant improvement over previous systems, such as Munin, CVM and Disom [9, 17, 10] etc., none of which provide all the flexibility of the system described here.

The implementation and performance sections present proof of concept implementations based on a small set of consistency models and coherency protocols. Using these we demonstrate the improvements in performance to be accrued from tailoring the combinations of consistency models and coherency protocols to suit the application's semantics.

Acknowledgements

Stefan Weber is supported by a basic research grant from Enterprise Ireland.

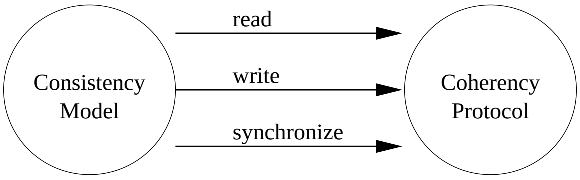

Figure 1: Relationship Between Consistency Models And Coherency Protocols

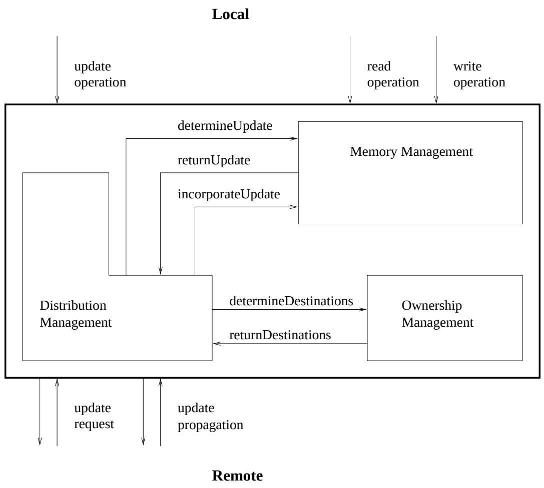

Figure 2: Coherency Protocol Overview

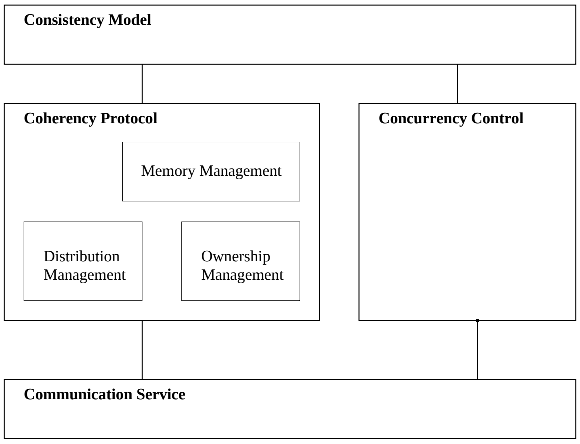

Figure 3: Framework Overview

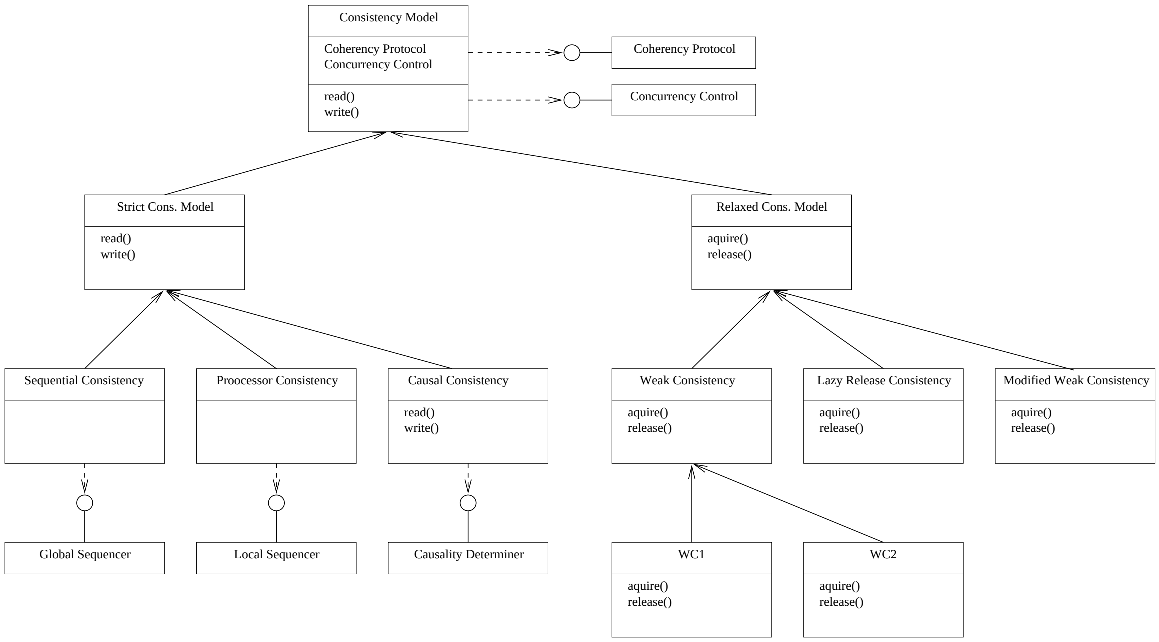

Figure 4: Inheritance Hierarchy For Classes Representing Consistency Models

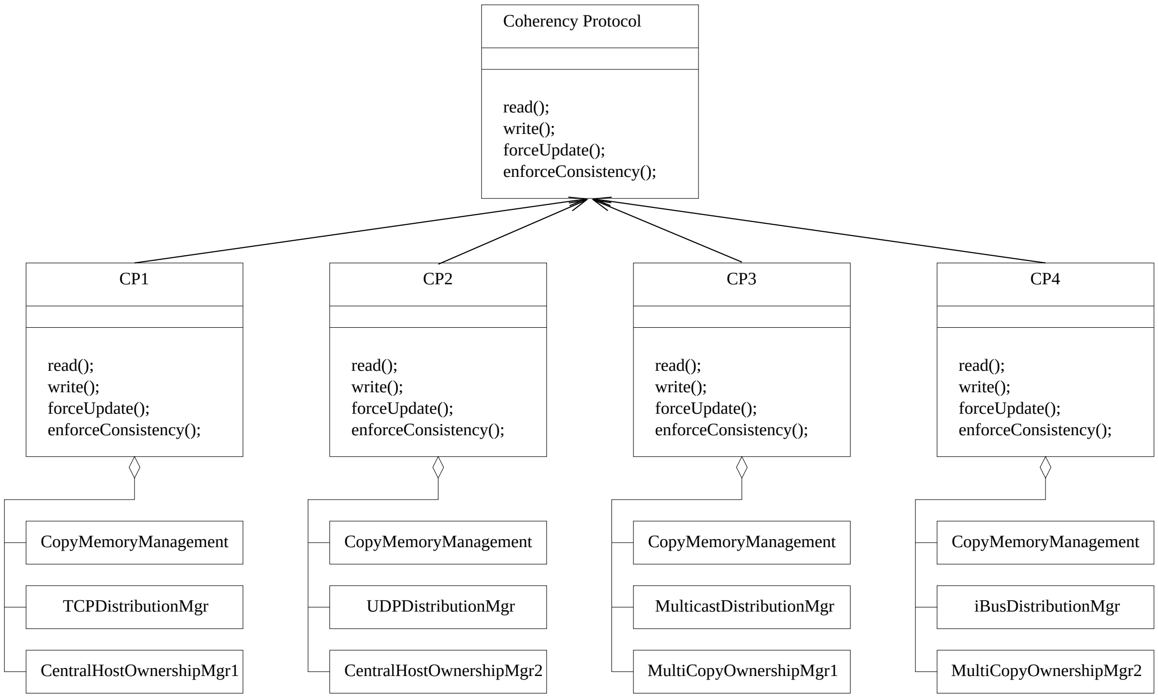

Figure 5: Inheritance Hierarchy For Classes Representing Coherency Protocols

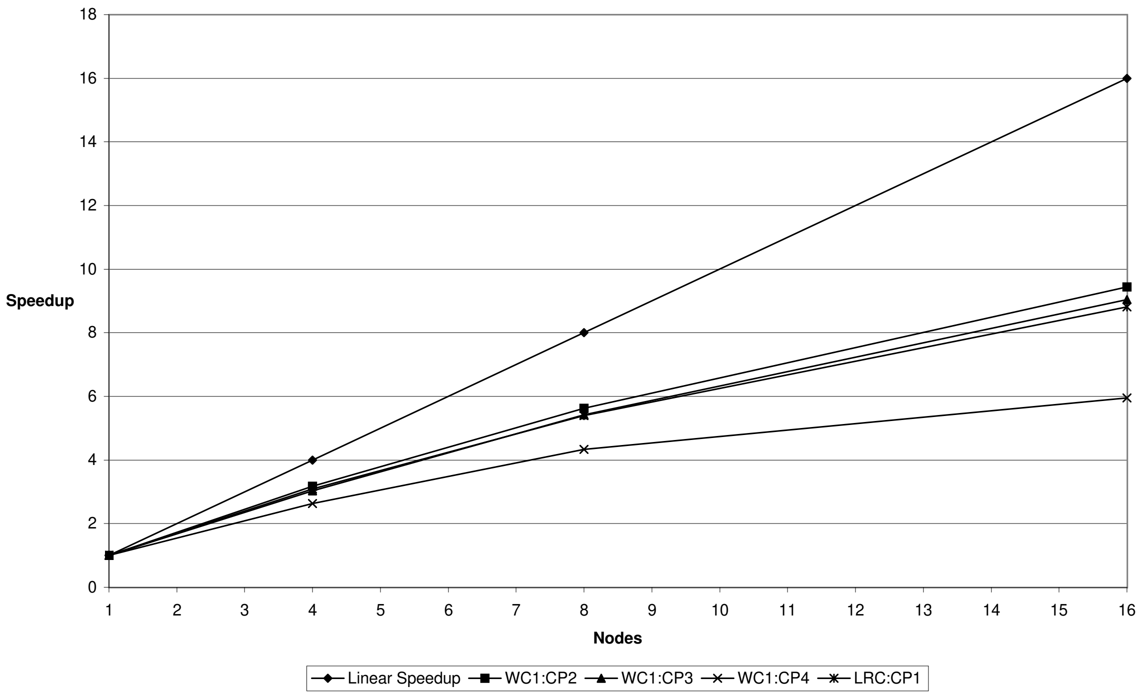

Figure 6: Matrix Multiplication Speed-up Curves

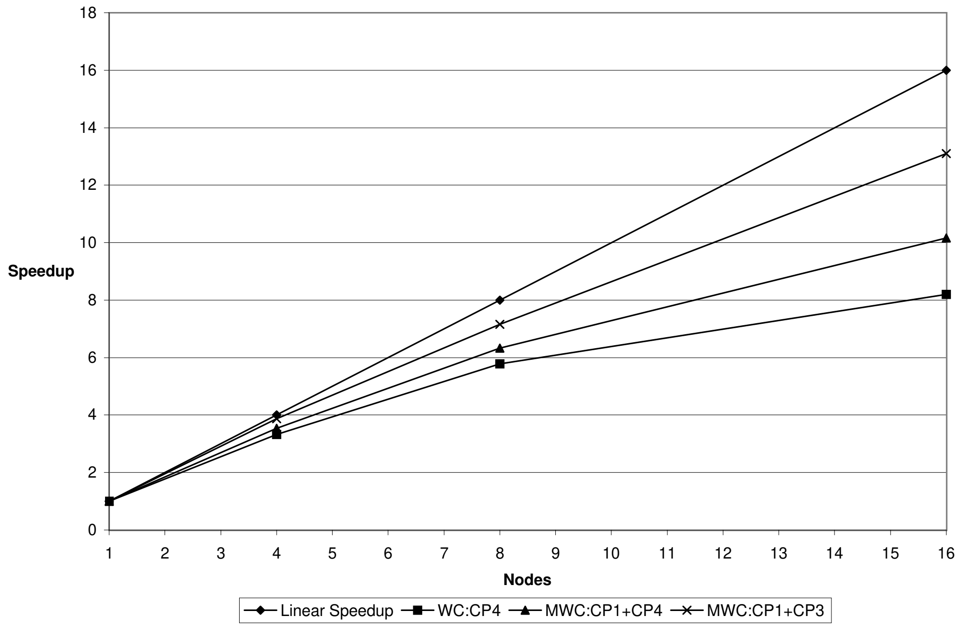

Figure 7: Life Speed-up Curves

References

[1] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[2] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. A distributed implementation of the shared data-object model. In *Proc. of the USENIX Workshop on Experiences with Distributed and Multiprocessor Systems*, pages 1–19, October 1989.

[3] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Adaptive software cache management for distributed shared memory architectures. In *Proc. of the 17th Annual Int'l Symp. on Computer Architecture (ISCA'90)*, pages 125–135, May 1990.

[4] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proc. of the Second ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP'90)*, pages 168–177, March 1990.

[5] B. N. Bershad and M. J. Zekauskas. Shared memory parallel programming with entry consistency for distributed memory multiprocessors. Technical Report CMU-CS-91-170, School of Computer Science, Carnegie-Mellon University, September 1991.

[6] Grady Booch. *Object-Oriented Analysis and Design with Applications, 2nd Ed.* Benjamin/Cummings Publishing Company, Inc., 1994.

[7] H. Burkhardt, S. Frank, B. Knobe, and J. Rothnie. Overview of the ksr1 computer system. Technical Report KSR-TR-9202001, Kendall Square Research, February 1992.

[8] R. Campbell and N. Islam. Choices: A Parallel Object-Oriented Operating System. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming.* MIT Press, 1992.

[9] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and performance of munin. In *Proc. of the 13th ACM Symp. on Operating Systems Principles (SOSP-13)*, pages 152–164, October 1991.

[10] M. Castro, P. Guedes, M. Sequeira, and M. Costa. Efficient and flexible object sharing. In *Proc. of the 1996 Int'l Conf. on Parallel Processing (ICPP'96)*, volume 1, pages 128–137, August 1996.

[11] P. Dechamboux, D. Hagimont, J. Mossière, and X. Rousset de Pina. The Arias Distributed Shared Memory: an Overview. In *Proc. of the 1996 23rd Seminar on Current Trends in Theory and Practice of Informatics*. INRIA, November 1996.

[12] M. Gardner. Mathematical Games: The fantastic combinations of john conway's new solitaire game 'Life'. *Scientific American*, 223(4):120–123, October 1970.

[13] K. Gharachorloo, S. V. Adve, A. Gupta, J. L. Hennessy, and M. D. Hill. Specifying system requirements for memory consistency models. Technical Report CSL-TR-93-594, Computer Systems Laboratory, Stanford University, December 1993.

[14] P. Guedes and M. Castro. Distributed shared object memory. In *Proc. of the 4th Workshop on Workstation Operating Systems (WWOS-IV)*, pages 142–149, October 1993.

[15] Alan Judge. *Supporting application-consistent distributed shared objects*. PhD thesis, Department of Computer Science, Trinity College, October 1996.

[16] P. Keleher. The relative importance of concurrent writers and weak consistency models. In *Proc. of the 16th Int'l Conf. on Distributed Computing Systems (ICDCS-16)*, pages 91–98, May 1996.

[17] P. Keleher and C-W. Tseng. Enhancing software dsms for compiler-parallelized applications. In *Proc. of the 11th Int'l Parallel Processing Symp. (IPPS'97)*, April 1997.

[18] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[19] K. Li. Ivy: A shared virtual memory system for parallel computing. In *Proc. of the 1988 Int'l Conf. on Parallel Processing (ICPP'88)*, volume II, pages 94–101, August 1988.

[20] M. Mizuno, M. Raynal, G. Singh, and M. L. Neilsen. An efficient implementation of sequentially consistent distributed shared memories. In *Proc. of the IFIP WG10.3 Int'l Conf. on Applications of Parallel and Distributed Computing*, pages 145–154, April 1994.

[21] W. Shi, W. Hu, and Z. Tang. An interaction of coherence protocols and memory consistency models in dsm systems. *ACM Operating Systems Review*, 31(4):41–54, October 1997.

[22] SoftWired. ibus programmer's manual, August 1998.

[23] Yasuhiko Yokote. The Apertos Reflective Operating System: The Concept and Its Implementation. In *Proc. of the 1992 7th Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1992.

Stefan Weber received a Diplom(FH) in Praktischer Informatik from FHO Emden in 1995. He is a Ph.D. Candidate in the Department of Computer Science at Trinity College Dublin. His research interests focus on distributed shared memory systems and component-based object-oriented systems.

Paddy Nixon B.Sc(Liv) 1990, Ph.D. (Shef) 1994, C.Eng, is a lecturer in the Department of Computer Science in Trinity College Dublin. His research are in distributed systems, mobile computing, and virtual enterprises. Dr. Nixon has been a visiting scholar at Kaunas Technical University, Lithuania, Warsaw University, Poland and California Institute of Technology, USA.

Brendan Tangney B.Sc.(N.U.I.) 1980, MSc (Dub) 1982, is a lecturer in the Department of Computer Science in Trinity College Dublin. His research interests are in distributed systems, cluster computing and distributed virtual environments. In 1990 he was a guest scholar in the Department of Information Science in Kyoto University, Japan.