Meta-level architecture support for distributed objects

Jeff McAffer

Department of Information Science
The University of Tokyo
and
Object Technology International
`jeff@acm.org`

Abstract

The creation of distributed applications is often hindered by a lack of a priori knowledge of distributed object behaviour. Designers and builders of distributed systems can benefit from an environment which allows them to explore and experiment with various computational and structural models for application objects. Our ability to do this is limited by traditional distributed systems' tendency to mix domain-specific object description and distributed behaviour specification. By using an explicit meta-level architecture, we can transparently add a wide variety of distributed behaviours to objects with little effect on their base-level behaviour or code. In this paper we outline such an architecture and detail the design of various distribution mechanisms and policies (e.g., marshaling and replication). These are shown to be independent of base-object behaviour to such a degree that they can be recursively applied to the architecture in which they are implemented.

1 Introduction

The creation of distributed applications is often hindered by a lack of a priori knowledge of how objects will react to distribution. The use of strongly typed languages can help static analysis techniques determine call graphs and interaction patterns but it is difficult to account for the dynamic nature of distributed systems (e.g., the same application may run differently depending on the machine topology).

In our approach, the separation of the base-level application code from the meta-level object behaviour (e.g., distribution) code plays a key role. This separation enables distributed application developers to prototype their applications and experiment with distribution models while minimizing the effects on the application's code. Normal objects are reused in a distributed environment by transparently adding distribution behaviour to their meta-level. In addition, new distribution mechanisms are more easily integrated with existing object behaviours.

We have developed *Tj*, a platform for describing distributed object behaviour. *Tj* is built on top of CodA[11], a general meta-level architecture which opens the implementation of an object's fundamental computational behaviours (e.g., state accessing, message sending, queuing, method execution). This power is required because describing distributed object behaviour is more than just implementing remote references. *Tj*'s DistributedObject model contains comprehensive notions of object spaces, machines, topologies, remote references, object marshaling, replication and migration. These mechanisms are implemented in an open and extensible way so as to accommodate user changes and additions. The mechanisms themselves are reified as meta-level *components* and provide a place to define both their implementation and their use (i.e., policies).

Particular emphasis is placed on argument and return value passing techniques (i.e., *marshaling*). While most systems provide a means of specifying marshaling on a per-object or per-object group (e.g., class) basis, this is not enough. Objects are often used simultaneously in many different contexts. We must be able to specify marshaling on a per-use basis. *Tj* provides an open, extensible marshaling framework based on declarative *marshaling descriptors* which are specified by users or by the system via automatic analysis.

This paper deals mostly with design topics and leave implementation and performance issues to a companion paper which is in preparation. The remainder of this paper is organized as follows. Section 2 outlines the CodA meta-level architecture while section 3 details our proposed distributed object model. A further section describes object mobility within our model and section 5 gives an example of object replication. Finally, in section 6 we draw some conclusions and discuss possibilities for future work.

2 The CodA Meta-level Architecture

CodA[1] is a general-purpose, extensible framework which opens or exposes various aspects of an object's implementation. As such it is part of the emerging field of *open implementations*[13] with other work such

[1]We give a brief sketch here and refer interested readers to a fuller description in [11]

as; CLOS MOP [9], ABCL/R2 [10] and RbCl [7] to name but a few.

The key goal in all of this work is to separate *what* and object does (its *base-level*) from *how* it does it (its *meta-level*). Having done this, we can alter the computational characteristics of objects in isolation of their base-level semantics. Sequential objects can be run in concurrent environments, state storage strategies can be altered etc. All without changing the main code of the object. In this paper we present the application of the CodA architecture to the area of distributed computing.

In CodA, the meta-level is decomposed into a number of *facets* corresponding to particular areas of object behaviour such as; execution (both mechanisms and resources), message passing, message to method mapping and state maintenance. Each facet is filled or described by a *meta-component*. Facets may be filled by one of many different components and one component can describe several facets. An object's behaviour is changed by explicitly redefining components or by extending the meta-level's set of facets.

Components are typically small and simple and define a clear *programmer's interface* (what some might call a Meta-Object Protocol or MOP). They represent a fine-grained decomposition of object behaviour. An object's behaviour is modified by adding or substituting different components for the relevant facets of its meta-level. For example, if we would like to change the way an object looks up messages (e.g., inheritance), we change its Protocol component.

If the current component supports some sort of parameterization, it may be enough to simply modify it's settings. If not, we substitute a completely new Protocol object. If we need to describe a completely new behaviour then we can add facets to the meta-level and implement new components to fill them. It should be noted that not all combinations of components are useful or possible.

Sets of meta-components are grouped together into *object models* or higher-level descriptions of object behaviour (e.g., concurrent or distributed objects). They form consistent, coherent wholes which can be manipulated as a unit.

As a basis, CodA defines a default set of seven components which are present for all objects.

Send Defines how an object handles outgoing messages. This includes protocol negotiation and synchronization.

Accept Implements the external interface for incoming messages. Accepts interact with the sending object's Send to determine if a message is valid and concerns the base-object. Note that this is different from receiving.

Queue Organizes and holds messages which have been accepted but not yet received or processed. Queues are the main mechanism for decoupling sender and receiver.

Receive Defines the operations related to fetching accepted and queued messages for processing. To receive a message is to consider processing it. Even though a message has been accepted by the Accept, it can later be ignored by the Receive. When asked for the next message to process, a Receive may consider messages from many queues and various synchronization constraints before taking a particular message.

Protocol Maps messages onto methods for execution using some scheme (e.g., inheritance).

Execution Describes how an object executes methods both in terms of the execution model and the processing resources.

State Defines what state an object has and how that state is stored and retrieved. Note that a State *does not* actually hold the data, it simply knows how it can be accessed.

2.1 Example Meta-level

In Figure 1 we depict the events, meta-components and interactions involved in the sending of a message M from object A to object B (as indicated by the heavy dashed arrow). The shaded areas contain meta-components. Each light arrow is an interaction event (dashed for A's execution thread, solid for B's). The heavy solid arrows indicate the base/meta relationship and go from base-level to meta-level. We have labeled only those meta-components relevant to this particular interaction.

We see that A sends M by interacting with it's Send (1). The Send then transfers M to B's Accept (2) which queues it with the Queue (3). At some point, B will execute a receive operation which invokes the Receive (4) and fetches the next message from the Queue (5). The message is mapped to a method by the Protocol (6) and finally, the message is processed by executing the found method (7). In this way, every aspect of basic execution is reified.

2.2 Implementation

CodA has been implemented in Smalltalk and much of the remaining discussion draws from that experience. Because CodA deals largely with execution rather than language issues, we have been able to fully integrate it into Smalltalk. For each Smalltalk object we transparently add the meta-level infrastructure and default meta-components which describe the standard Smalltalk object model — A standard CodA object behaves just like a standard Smalltalk object. Having opened its implementation, we can adjust the object's behaviour as needed. Objects generally retains its base-level semantics but gains some additional behaviour such as concurrency or distribution.

3 The Distributed Object Model

There has been relatively little work done using the meta-level for implementing distributed object systems. AL-1/D [12], Apertos [16] and GARF [5] being notable exceptions. As systems get more complex, diverse and dynamic, issues of object behaviour become more and more important. We must be able to describe objects which move and adapt in many ways.

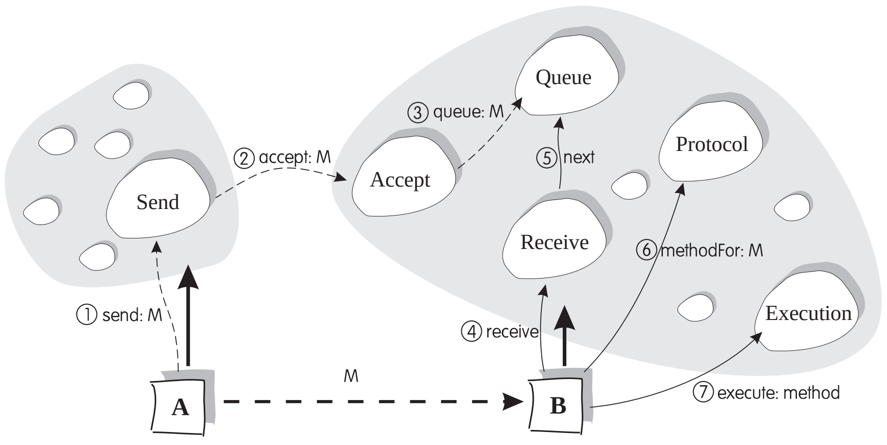

Figure 1: Sample meta-level configuration and interaction

Meta-level architectures give us "a place to stand" and describe both the mechanisms for implementing this behaviour and the policies for their use. Having a clear separation between the base and meta-levels isolates unrelated, application domain (i.e., base-level) code from the dynamic behavioural (meta-level) code. Objects can be reused in a variety of computational environments with little or no change and common object behaviour descriptions can be used on a vast array of domain objects.

Using the meta-level facilities provided by CodA, we have implemented a distributed object system called *Tj*. *Tj* provides both a distributed computation model with remote references, replication, migration, etc. and a set of distributed system infrastructure objects. A *Tj* distributed system is a collection of DistributedObjects living in *object spaces* running on *machines* and interconnected by *transport mechanisms* in some *system topology*.

Our approach differs from existing systems in the following ways. The GARF architecture, while providing much of the functionality required for distributed systems, is limited to two levels, base and meta and the domain of distribution. That is, it is not based on a fully general meta-level architecture.

AL-1/D uses a somewhat coarser (compared to CodA) decomposition of the meta-level into *models*. The bulk of CodA's standard components would make up just one of AL-1/D's models (the Operation Model). Furthermore, the decomposition is not balanced. Some models are large and significant (e.g., Operation) while others are small and contain little functionality (e.g., Statistics). There also appears to be no general framework for managing the models.

Apertos has many good features but is somewhat lower-level than our work. This is a result of their focus on operating system issues. For example, while their meta-level is concerned with contexts, memory page faults and the like, CodA/*Tj* deals in message sends, slot accesses etc, in a sense, object OS issues. We do not claim that one approach is superior to the other. They are somewhat orthogonal and are both required for completely open distributed computing.

SOS and its Fragmented Objects[14] is also related to our work despite its lack of an explicit meta-level or framework for meta-level concept description. The ability to transparently 'fragment' objects over processors smacks of meta-level behaviour description. It was through looking at these issues that we discovered one of the most important properties of our distributed system; the reification of an object's state and execution behaviours into distinct objects.

This property is a direct consequence of the CodA object model's explicit definition of the State and Execution meta-components. In the distributed domain it means that an object can store its state in one space but execute in another. Extending this property, an object's state and execution can migrate independently. Having enabled this, it is a simple step to fragment the actual state storage over multiple object spaces. Such relationships and properties are exploited throughout *Tj*'s distribution mechanisms.

3.1 RemoteReference

Also fundamental to the distributed architecture is the RemoteReference. A RemoteReference is a local representative or *Proxy*[14] for some remote object. Locally they are just like any other object. They can be stored in instance slots, assigned to variables, passed as arguments, etc. When sent a message, the simplest RemoteReference just forwards it to the space containing the real object — its *target*. More sophisticated RemoteReferences process some messages locally while forwarding others to the target.

RemoteReferences are themselves implemented using modified CodA meta-components. According to the CodA execution model, when a message is sent to an object, the sender's Send and the receiver's Accept interact to effect the message transfer. In the distributed case, these meta-components are in different spaces. Local to the sender, the receiver is a RemoteReference and its Accept is an intelligent RemoteReference to the target's Accept.

Rather than performing the normal accept operation, the local Accept *marshals* the message into a stream of bytes and transmits it to the remote space. Once there, it is reconstructed and accepted by the real Accept. In this way, the DistributedObject model is uniformly applied to all objects in the system, even to those of the meta-level architecture in which it is implemented!

3.2 Execution model

In general, the execution model of *Tj* objects is either *active* or *passive*. An active object has a thread of its own while passive objects do not. Normally, when a passive object receives a message, it *borrows* the thread of the sender method for the duration of the corresponding method execution. However, when a message is received from remote object, there is no local thread to borrow. The system provides a thread which does not belong to a particular object but simply executes a set of message sends from some root method. When that method exits, the thread dies or can be reused.

Active objects can have *active methods*. An active method is a method which is executed exclusively by the object's thread. On the other hand, *passive methods* can be executed by any thread. These facilities are used largely in support of controlling concurrency and synchronization in a way similar to Emerald's monitors [8]. CodA also supports more sophisticated intra-object synchronization mechanisms but these are not discussed here.

3.2.1 Parallelism

Distribution is introduced into a system for a number of reasons; because it fits the problem domain, for fault tolerance or for increased performance. We can gain performance through concurrent or parallel execution of distributed objects. To describe parallelism within a distributed object, *Tj* includes notions of *distributed message arrival* and *distributed method execution*.

Distributed message arrival is the idea that when a message arrives at some object, it is simultaneously distributed to all versions (e.g., replicas) of the object. This is different from traditional message multicasting in that it is defined by the receiver rather than sender. Distributed method execution implies that when a method is invoked (i.e., actually run), it is invoked for all versions (e.g., replicas) of the object.

These notions are different in two ways. First, message arrival does not imply the execution of a particular (or any) method. This mapping is determined by the receiver's meta-level. Second, since messages may be queued before being processed, message arrival and the method execution which may follow are temporally decoupled events.

3.3 Marshaling

Marshaling, or the packaging of objects for inter-space transmission, can have a profound effect on the expressiveness and efficiency of a distributed system. The *Tj* marshaling scheme allows programmers to specify object form in a clear and simple way but is sophisticated enough to handle complex behaviours.

Marshaling is required for passing receivers and parameters in message sends and return values in message replies. In *Tj*, an object's marshaling policies are defined at the meta-level by the component filling the newly created Marshaling facet. These policy descriptions are an interface to the general marshaling mechanisms supplied by *Tj*.

3.3.1 Marshaling mechanisms

To build a distributed system, it is technically sufficient to supply just a pass-by-reference mechanism. Unfortunately, the exclusive use of referencing leads to a dramatic increase in cross-space references and messages. This in turn leads to a decrease in performance of both user code and system code (e.g., distributed GC).

Passing parameters by value (i.e., by copying) reduces cross-space messages but at the expense of increased message size. There is also a loss of generality as copying is typically only applicable to *immutable* objects where it will not affect semantics. Work with Emerald [8] has explored more sophisticated techniques such as pass-by-move and pass-by-visit and found them to be useful.

We have developed a generalized marshaling mechanism based on the notion of *marshaling descriptors*. These descriptors give *hints* as to how an object should be marshaled. Examples are: reference, shallow/deep copy, replica, etc.

An object's Marshaling defines a default descriptor which is used if no other descriptor is specified or derived. In general, user-supplied descriptors can override a default however it is possible to prevent or constrain this. Descriptors are also *descriptor generators* in that they produce descriptors for the various parts of the object whose marshaling they describe.

At the heart of the marshaling mechanism is a generic object graph walker or *marshaler*. The marshaler walks object graphs according to a series of marshal descriptors. At each object in the graph it invokes the operations specified by corresponding descriptor. The marshaler maintains the minimum desired, current and maximum desired traversal depths as well other *global* information such as a marshaled object registry used for cutting cycles. These combine to give descriptors a global view of the marshaling process for use in determining how to proceed.

In addition to several low-level system descriptors, there are some ten to fifteen different ways to marshal an object. A partial list of these is given below. This set is completely extensible allowing users to add new marshaling techniques in support of new distributed object behaviours.

Value Substitute some value held by the descriptor for the actual object being marshaled. Marshal the value according to a descriptor also held internally.

Basic Marshal the object's instance variables according to its contents' default descriptor.

Depth Traverse the object graph from the current object to a minimum and maximum depth as specified by the descriptor. Using this mechanism we can specify an infinite range from shallow to deep copy.

Reference Marshal a global reference to the object.

Cached Reference Marshal a global reference such that the first time it is accessed, the reference is resolved locally according to a descriptor held by the reference. Note that this resolution descriptor can take any form.

Replica Replicate the object in the receiver's space. The object is replicated according to a further descriptor held internally.

Move Move the object to the receiver's space. The object is moved according to a further descriptor held internally.

Use Consult the receiver to determine the marshaling form which best suits the use of the object. Having invoked some sort of automatic analysis system to analyze method argument use, objects can export this information via their Accepts. Objects sending messages then determine, before the message is sent, how the arguments will (may) be used and marshal them accordingly.

Attach Transparently marshal and thus transport a given set of objects with the annotated object.

Descriptors for these attached objects may also be specified. This is equivalent to object attachment as seen in Emerald (see below).

Slot Specify, on a per-slot basis, the inclusion or exclusion of slot contents and the descriptors to use in their marshaling.

Operation Specify a block of code to be used in marshaling the object. This is the escape mechanism which enables arbitrarily complex marshaling.

Descriptors can be computed or declared. Computed descriptors are often the result of some analysis process while declared descriptors occur as annotations to messaging operations. A message annotated with a declarative marshaling descriptor is shown below. Note that computed and supplied descriptors are handled in the same way by the system. They are just derived and attached differently.

```
someObject <-
    foo: arg1 {deep}
    bar: arg2 {replica}
```

The above sequence sends the foo:bar: message to someObject and marshals the first argument using deep copy. The second argument is replicated in the receiving space. In this case it is replicated according to the arg2's default marshaling descriptor.

If we wish to specify how the replication will take place (i.e., the form of the replicas), we can specify a further, *nested*, descriptor such as, {replica: (-3 20)}. This specifies that the argument is passed as a replica which is a traversal of the argument's object graph to a minimum of depth 3 and a maximum of depth 20.

This notion of nesting descriptors can be applied in many situations. For example, when using cached reference marshaling we also specify a descriptor to be used to resolve the remote reference when it is located. We may even choose to resolve the reference with a replica (e.g., {cached: replica}). Variations on this nesting theme can be as complex as required and can involve almost any of the descriptors mentioned above.

3.3.2 Policies

A rich, extensible set of marshaling mechanisms is only part of the answer. We must also be intelligent about how we determine which mechanism to use for a particular object in a particular situation. The optimal strategy can be determined by looking at user-supplied information, the objects (arguments) themselves or the message being sent. This is the role of the Marshaling component.

An object with no marshaling descriptor annotations uses the defaults supplied by its Marshaling. The default is generally determined by some internal property (e.g., class or type). For example, most systems pass objects as references unless they are immutable, in which case they are passed as values. *Tj* objects generally follow this model.

Systems like Emerald support object-specific marshaling specification using object *attachment*. Under the attachment mechanism, users explicitly attach objects to one another. When an object is copied, all of its attached objects are copied. *Tj* supports this in the form of attach marshaling descriptors. It also supports slot-based attachment. Under this model, marshaling descriptors specify the attachment of objects by virtue of the slot in which they are held (see slot descriptor).

Many systems have a uniform view of marshaling on all instances of a particular type of object — all objects of type X are passed using the same technique(s). Other systems allow the specification of marshaling on a per-object basis (e.g., object attachments). In contrast, the *Tj* marshaling system supports descriptor specification on a *per-use* basis. This is important in realizing optimizations and system-level mechanisms.

3.3.3 Analysis and Optimization

The automatic optimization of user-level code depends largely on the analysis of particular sends to determine how parameters and return values should be passed — It is, by definition, use-case specific.

Systems which support use-case marshaling specification, generally do so via automatic analysis techniques. Static analysis with strong-typing can determine a method's marshaling requirements quite accurately by looking at its parameter and return value types and references, and examining the call-graph. Systems like Orca [1, 15] and Munin [3] use such techniques. Essentially they treat strong-typing information as variable use-case information.

The situation is not so good for untyped systems like Smalltalk. Unfortunately, classes (or even types) are not sufficient to enable analysis or runtime systems to determine the best or correct marshaling strategy. This is particularly true of polymorphic and system-level code (e.g., the implementation of a replication mechanism).

Polymorphic variables default to the most general type or class of all possible uses. As such, structural information essential to the marshaling analysis is removed from the system.

System-level code often functions 'underneath' objects and so should not be subject to the same abstractions as normal object clients. In these situations, marshaling must be specified in a way which is independent of the objects' normal behaviour and specific to the particular use. These specifications can come from static programmer declarations or be dynamically derived.

For example, suppose we developed a metric for determining how many times a method argument is accessed. Using this metric we might suggest that arguments referenced more than X times be passed by value or by migration so they are local. On the other hand, at run-time we may find that one of those arguments is in fact a very large structure and that copying or moving of the object would be costly. Clearly there is a trade-off. Note that strong typing does not address this case as the copied size of an object is determined by its type *and* things like its attachments which may be dynamically determined.

We propose the use of run-time *negotiation* to weigh the accessing costs against the copying/movement costs and determine the appropriate action. By negotiation we do not mean some heavy-weight, multi-iteration conversation between the sender and receiver over the processor inter-connect channels. Rather, we use local meta-level information.

By giving out a reference to itself, an object is projecting a part of its Accept into the remote space. The sender's Send component can communicate and cooperate locally with the receiver's projected Accept to determine the best communication strategy.

By implementing *Tj* using a rich meta-level environment (i.e., CodA), we expose implementation information and allow the system to make informed choices. This exposure does not violate encapsulation because it is the object's choice to provide information. *Tj* provides the mechanisms and framework for using the available information to calculate declarative marshaling specifications which suit the needs of a particular object interaction. The mechanisms scale to handle a full range of marshaling descriptors from broad hints to precise forced specifications.

4 Object mobility

As discussed in the overview of *Tj*'s distributed object model, an object's state and execution are independent. As such, the issues related to the mobility of their state and execution are somewhat orthogonal. In fact, since an object's Execution is just another object, the implementation of its mobility is largely the same as that of any other object. To control the mobility of an object, we create two new facets for the meta-level; Replication and Migration.

4.1 Execution mobility

In terms of mobility, we do not consider the possibility of Execution replication since replication generally comes with some level of global consistency. *Tj* assumes a MIMD computing model and expects the various copies of an object's Execution to evolve independently making global consistency undesirable.

Instead, an object's Execution can be copied or migrated to remote spaces in a way similar to the computational migration seen in [6]. Note that an object can have Executions in many spaces and still only maintain one version of its state. This flexibility is a direct consequence of the separation of the State and Execution behaviours of objects and gives rise to the following six possible relationships between versions of an object's State and its Execution:

$1:0$ Normal passive object. This is the default case. Objects have one state location and borrow threads from their senders.

$1:1$ Normal active object. Objects have one state location and a dedicated thread for their execution. These may be remote from each other.

$1:N$ Parallel object with single state. Useful where the object makes many accesses to remote objects and few to itself. Simply copy the Execution to all the relevant spaces and the methods execute locally. Optional method distribution will result in all copies of the Execution executing the same method though not necessarily in lock-step. State accesses are serialized through the one copy of the state.

$M:0$ Replicated passive object. Each replica is passive and borrows threads local to its state. This results in a degree of implicit parallelism. State accesses are controlled according to the replication consistency model.

$M:1$ Replicated active object. Use cases for this may seem somewhat contrived but they are nonetheless feasible as potential behaviours. Consider the case where a particular object is large, as is the number of references it does to itself and the objects in any given space. The object's execution is such that it processes data in one space discretely and then moves to the next. In this case, it may be more efficient to replicate the state once in each space. Then we can avoid the iterative state migration and execution synchronization cost by migrating a single Execution among the replicas.

$M:N$ Replicated parallel object. This is a mixture of the above models.

Methods for an object executing remotely, do so such that the receiver (i.e., self) is a RemoteReference to the object which is the master copy of the object's state. All instance variable accesses are converted to remote message sends to the nearest space which maintains state for the object. Depending on how the Execution was created, the local version of the object may also maintain local versions of the object's other meta-components such that some messages can be handled locally.

Tj does not support full thread migration in the sense that arbitrary threads cannot be migrated at arbitrary points in time. There is no design limitation imposing this restriction, it simply has yet to be implemented. As it is, migration (and in fact copying) can only be done on message processing boundaries. This is to avoid the need to copy and recreate portions of the stack. Since we are on a message boundary, we can simply construct the appropriate stack base for the execution in the new space. There are some outstanding technical issues relating to the migration of active objects which will be addressed in future work as the need arises.

4.2 State Replication

When an object's state is replicated, the *master* (the original) is copied to one or more remote spaces creating a number of *clients*. The master's meta-level is modified such that state changes are trapped by the Replication component. Replication is in fact independent of state form and can accommodate radically different representations in different spaces. Replication components themselves are quite simple.

The master's Replication maintains a list of spaces which contain replicas. On state change, it coordinates with those spaces to update the clients according to some consistency model. The clients' meta-levels contain a counterpart Replication component which has agreed to an update protocol and knows the identity of the master object. Depending on the consistency model, state reads are also routed through the Replication.

The replication model for an object is specified via a descriptor. In section 3 we presented a subset of the marshaling descriptors available in *Tj*. Readers will note the presence of a replica marshaling descriptor. This uniform mechanism admits the following possibilities:

- Parameter passing by replication. This is a novel mechanism in which the argument is copied with an indication that when it is received in a space, it should report back to the master for consistency management.
- Varying consistency models by slot. Allowing different consistency models recognizes the fact that objects may have different use patterns from application to application and that the demands they place on their state variables may not be homogeneous.
- Slot form specification. While the consistency model describes how and when replicas are updated, specifying the form is directly analogous specifying the parameter marshaling descriptor used when updating a remote value. In message passing, a use-case may demand that a particular slot of an object be passed in a certain form (e.g., copied to a particular depth). In replication we may encounter the same use-case and would expect that slot of the object be replicated in the same way (i.e., as a copy to a particular depth). Here we allow the full range of marshaling mechanisms for use in specifying remote slot form. Replicated slots can even be updated by further replicating the value in the slot!

4.3 State Migration

The mechanisms for State migration are quite similar to those for replication. Migration is essentially a copy operation followed by a global pointer update. It does not place any "after operation demands" on the originating space other than the need for a migrated object location mechanism [4].

As with replication, objects are migrated according to the specifications given in the supplied descriptor. An object can be migrated in almost any form. For example, specifying {move} for a message argument or object slot invokes the default migration. {move: descSpec} moves the object in the form given by descSpec, another descriptor. {cache: move} describes a *cached reference* which will move an object only if it is accessed in the remote space.

There are also some issues related to the threads executing in an object when it is migrated. By in large, these are handled by the local object replacement mechanism. When an object's state is migrated, the local version of the object is replaced by a remote reference to the object in its new location. As such, any references or messages to the object will be forwarded to its new location.

In addition to specifying the form of the migration, the Migration also defines the policies for migration. For example, where and when the object is to be migrated. In many cases, the user/programmer may *hint* that an object should be migrated. Say in a parameter passing marshal descriptor. It is then up to the parameter's Migration to provide additional information (e.g., cost, size, complexity) in support of the parameter passing negotiation techniques discussed above. This is also true of Replications.

5 Replication Example

To demonstrate replication we develop the partial replication scenario shown in Figure 2. The figure shows two objects, original (in space 0) and replica (in space 1). Though not shown, original is actually a 2D N-Body [2] problem solver which calculates the forces exerted by, and movements of, a collection of bodies or *particles* in a 2D plane. N-Body solvers arrange a set of particles in a Quad tree structure according to their physical location and then process each particle individually. Overall, processing consists of a couple tree scans and iterations over the collection of particles.

To distribute this algorithm we divide the particles into subsets which are worked by different solvers, one

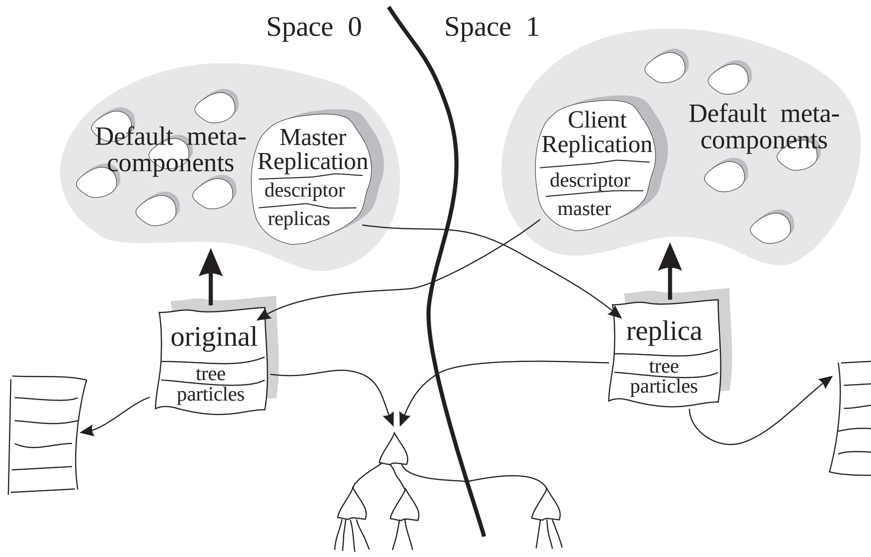

Figure 2: Distributed object layout

per space. The sets however, are not entirely independent as all particles potentially exert forces on all others. The tree is the central data structure for relating particles to one another and must be globally known and unique. Here we detail the partial replication of the solver.

As shown in Figure 2, original, the solver, has two slots; particles and tree. replica is a partial replica of original where the tree slot is consistency managed and the particles slot is not. All the replicas in the system share the same tree but have independent particle sets. The replication of original is done in a series of six steps. Figure 3 shows the required code while the discussion below explains each step.

1. Ensure that the original's Replication compatible with the behaviour described by the MasterReplication component. It should be able to detect state changes in the appropriate slots and maintain a list of replicas. The first argument is a marshaling descriptor which specifies how the slots of original are to be copied to the remote space and as a result, how original is to be replicated. Simply giving a slot name indicates that the slot is to be replicated using whatever marshaling technique is appropriate at the time (i.e., the default).

2. Invoke the replication operation and specify which spaces are to receive replicas. In keeping with our example, only space 1 is specified.

3. Copy the relevant slots of original to all of the specified spaces. The replicate:using:for: message has three arguments. Though the first and third appear redundant, they are not — they are marshaled differently. The first argument is marshaled according to the specification in descriptor while the third is marshaled as a reference. This difference is critical for the next two steps. When the message gets to the remote space, the first and third arguments will no longer be identical. The first will be a copy of base while the third will be a reference to base. Note that though marshaling descriptor specification is a simple addition to the messaging syntax, the details are omitted from this example to improve clarity.

4. Make the remote copy into a *client* of original. This is similar to step 1 and executes in the remote space which will contain replica. copy's meta-level is modified such that all state changes are delegated to master and its Replication knows the identity of its master for future reference.

5. Convert any preexisting remote references to original to be local references to replica. Remote spaces may contain references to master prior to replication. To maintain a consistent view of the world, these remote references should be changed into local references to the newly created replica.

6. Invoke consistency management on the replicated slots of original by adding the space to the list of consistency controlled replica locations.

In step 1 we hooked the relevant state change operations for original. Note that we do not require a new State component. The existing component's meta-level is manipulated to hook state accesses. This both isolates replication from representation and reduces the possibility of object model conflict. When original's replicated state is changed, its Replication's update:with:for method (shown below) is invoked by the hook. The method simply broadcasts the change in slot to all of original's clients.

```
1)  original meta replication asMasterUsing: #('tree') for: original.
2)  original meta replication replicateIn: (Spaces at: 1) for: original

MasterReplication>>replicateIn: space for: base
3)  space replicate: base using: descriptor for: base.
6)  replicaSpaces add: space

Space>>replicate: copy using: descriptor for: master
4)  copy meta replication asClientOf: master using: descriptor for: copy.
5)  master become: copy
```

Figure 3: The making of a replica

```
MasterReplication>>
   update: slot with: value for: base

   replicaSpaces do: [:space | | rep |
       rep := (base in: space).
       rep meta replication
          update: slot with: value for: rep]
```

In this example we have shown a relatively lax model of consistency. To implement *strict consistency* requires the addition of a two phase update protocol between masters and clients and the hooking or delegation of read state accesses on masters and clients as well as writes. Both of these changes are straightforward and are done using existing meta-level structures and mechanisms.

6 Conclusions and Future Work

We have detailed the DistributedObject model provided by *Tj*. It is a comprehensive distributed environment suitable for prototyping applications and experimenting with distribution mechanisms. *Tj* is based on the idea of object behaviour change at the meta-level. It uses the CodA meta-level architecture to open the implementation of objects and provide a framework in which to describe sophisticated distribution mechanisms such as marshaling, replication and migration.

The clear separation of base- and meta-levels facilitates the distribution of objects which were never intended to be distributed. In general, this requires very little change at the base-level. As such, programmers can reuse entire class libraries and experiment with quite different distribution models without major concern for base-level behaviour.

Tj provides a powerful framework for describing object marshaling on a per-object, per-object-group and per-use basis. Our model of marshaling descriptors presents the user/programmer with a single, clear and consistent model of interspace object transport specification. The descriptors are used both for parameter passing and mobility operations (e.g., replication and migration). They are uniform and recursive.

Overall, *Tj* addresses problems related to introducing distribution to systems which previously had none and in the description of complex distributed object behaviour. The framework itself is robust and extensible and we have found it to provide an excellent platform for application prototyping and behaviour experimentation.

The current version of *Tj* is implemented in Smalltalk and runs on the Fujitsu AP1000 MPP machine with up to 1024 nodes. It also runs on clusters of Unix workstations using MPI messaging. All of the mechanisms and models described here are implemented and running with several more in the design and implementation stage.

Future work in this area will be directed at the addition of distribution mechanisms (e.g., new replica coherency strategies) and automatic analysis of application code to direct the use of distributed mechanisms such as marshaling.

7 Acknowledgements

We gratefully thank Jean-Pierre Briot for his continuing interaction and helpful comments on drafts of this paper.

References

[1] Henri E. Bal and M. Frans Kaashoek. Object distribution in orca using compile-time and run-time techniques. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 162–177, September 1993. Published as ACM SIGPLAN Notices, volume 28, number 9.

[2] J. Barnes and P. Hut. A hierarchical $O(N \log N)$ force-calculation algorithm. *Nature*, 324:446–449, 1986.

[3] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 168–176, March 1990. Published as ACM SIGPLAN Notices, volume 25, number 3.

[4] R. J. Fowler. *Decentralized object finding using forwarding addresses.* PhD thesis, University of Washington, 1985. Also available as Dept. of Computer Science Tech Report 85-12-1.

[5] Benoit Garbinato, Rachid Guerraoui, and Karim R. Mazouni. Distributed programming in GARF. In *Proceedings of the ECOOP Workshop on Object-Based Distributed Programming*, LNCS 791, pages 225–239. Springer Verlag, July 1993.

[6] Wilson C. Hsieh, Paul Wang, and William E. Wiehl. Computation migration: Enhancing locality for distributed-memory parallel systems. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 239–248, July 1993. Published as ACM SIGPLAN Notices, volume 28, number 7.

[7] Yuuji Ichisugi, Satoshi Matsuoka, and Akinori Yonezawa. RbCl: A reflective object-oriented concurrent language without a run-time kernel. In *Proceedings of the International Workshop on Reflection and Meta-level Architectures*, pages 24–35, November 1992. Tokyo, Japan.

[8] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[9] Gregor Kiczales, Jim des Rivières, and Daniel Bobrow. *The Art of the Metaobject Protocol.* MIT Press, Cambridge, Massachusetts, 1991.

[10] Hidehiko Masuhara, Satoshi Matsuoka, Takuo Watanabe, and Akinori Yonezawa. Object-oriented concurrent reflective languages can be implemented efficiently. In *Proceedings OOPSLA '92, ACM SIGPLAN Notices*, pages 127–147, October 1992. Published as Proceedings OOPSLA '92, ACM SIGPLAN Notices, volume 27, number 10.

[11] Jeff McAffer. Meta-level programming with CodA. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, LNCS. Springer Verlag, August 1995.

[12] Hideaki Okamura and Yutaka Ishikawa. Object location control using meta-level programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, LNCS 821, pages 299–319. Springer Verlag, July 1994.

[13] The Open Implementations Workshop Participants. The open implementations workshop proposal and responses. Available on internet at: http://www.parc.xerox.com/OI/.

[14] Marc Shapiro, Yvon Gourhant, Sabine Habert, Laurence Mosseri, Michel Ruffin, and Celine Valot. SOS: An object-oriented operating system – Assessment and perspectives. *Computer Systems*, 2(4):287–337, Fall 1989.

[15] Andrew S. Tanenbaum, Henri E. Bal, and M. Frank Kaashock. Programming a distributed system using shared objects. In *Proceedings of the International Symposium on High Performance Distributed Computing*, pages 5–12, July 1993.

[16] Yasuhiko Yokote. The Apertos reflective operating system: The concept and its implementation. In *Proceedings OOPSLA '92, ACM SIGPLAN Notices*, pages 414–434, October 1992. Published as Proceedings OOPSLA '92, ACM SIGPLAN Notices, volume 27, number 10.