HEWLETT PACKARD

Problem-Oriented Object Memory: Customizing Consistency

Anders Kristensen, Colin Low
Intelligent Network Platform Department
HP Laboratories Bristol
HPL-95-85
July, 1995

relaxed consistency, object-orientation, distributed computing, problem-oriented protocols

This paper presents the notion of problem-oriented object memory, and its realization in a distributed object-based programming system, *Penumbra*. This system allows location transparent object invocation, object migration and caching. Its distinguishing feature, however, is its support for problem-oriented object sharing.

Problem-oriented object memory is an object model that allows exploitation of application specific semantics by relaxing strict consistency in favor of performance.

Our work addresses the problem of achieving scalability of shared write-intensive data in an environment of networked workstations. We have successfully applied the presented ideas to the management of a highly demanding telecoms application.

Internal Accession Date Only

Problem-Oriented Object Memory: Customizing Consistency

Anders Kristensen and Colin Low

Hewlett-Packard Laboratories, Bristol
Filton Road, Stoke Gifford
Bristol BS12 6QZ
United Kingdom
Email: {ak, cal}@hplb.hpl.hp.com

Abstract

This paper presents the notion of problem-oriented object memory, and its realization in a distributed object-based programming system, *Penumbra*. This system allows location transparent object invocation, object migration and caching. Its distinguishing feature, however, is its support for problem-oriented object sharing.

Problem-oriented object memory is an object model that allows exploitation of application specific semantics by relaxing strict consistency in favour of performance.

Our work addresses the problem of achieving scalability of shared write-intensive data in an environment of networked workstations. We have successfully applied the presented ideas to the management of a highly demanding telecoms application.

1 Introduction

We have investigated the issue of how to provide a high-level programming model for distributed applications which require very high-performance of a variety of types of shared data. Our approach is based on exploitation of application-level semantics to relax a traditional strict sense of consistency. This paper describes the abstractions we have designed to support such a model.

For a number of reasons the object-based approach to distributed programming has become the preferred one in recent years. We have carried out our work on problem-oriented consistency protocols in an object-oriented environment, namely that provided by the Penumbra toolkit. The result is a programming paradigm which we call *Problem-Oriented Object Memory* (POOM). This programming model allows relaxation of object consistency in favour of performance. Penumbra is presented, and the use of POOM is illustrated with examples from the telecoms world, especially from a network management application which we are currently developing.

One of the key points of this paper is that a distributed object-oriented programming system is well-suited as a starting point for implementing a generic framework for supporting relaxed consistency protocols. This is mainly due to the encapsulation features of OO systems.

1.1 The application: a distributed MIB

The application that has been the motivating factor behind our work is that of managing high performance telecommunications network elements (NEs) performing call-control. The important properties and requirements of this application are:

Performance. The NEs must be able to handle a very high number of service requests. We meet this requirement by functionally replicating the service logic, i.e. by processing multiple service requests in parallel over a distributed system.

Hierarchical MIB. The management information for a NE consists of a number of managed objects organized into a tree structure—the Management Information Base or MIB [13]. Due to replication, the MIB spans a

number of physical network nodes, but embodies a single logical database.

Single System View. For administration and management purposes, the NE MIB should appear to the network operator as a single system. It shouldn't be necessary, for example, to explicitly install new software on *every* node in the NE[1]. Thus parts of the MIB is shared between NE components and parts of it is local to individual replicas. The problem of providing a single system view is discussed in more detail in [19].

Access patterns. Managed objects exhibit different access patterns: some shared MIB objects have a high read/write ratio while other have a high ratio of writes to reads. This last category includes, for example, counters being updated on the order of incoming service requests. This suggests that different mechanisms are needed to meet the performance requirements of such objects.

Fault tolerance. Each NE has stringent availability requirements posed on it. Since algorithms exist for providing fault tolerance independently of the protocols and models discussed in this paper, we shall not consider fault tolerance any further.

The problem is thus primarily one of scalability: how to provide high performance of "hot" shared memory—data that is being updated with very high frequency by different processors.

Fortunately our management application has two properties which makes problem-oriented consistency a tractable approach. First managed objects are often rather simple, such as counters, and second clients are often able to tolerate or correct a certain level of inconsistency. Our work primarily addresses domains where these properties hold.

1. It must also be possible to manage nodes individually, but this involves no coordination between NE elements and is thus less troublesome to achieve.

1.2 Penumbra

Penumbra is a toolkit for writing distributed C++ programs. It is based on an early version of the *Shadows* toolkit from Newcastle University [7]. Penumbra is based on a principle of retaining C++ invocation syntax, semantics, and typing as far as possible and to provide support for POOM.

1.3 Organization of this paper

The rest of this paper is organized as follows. Section 2 introduces and justifies the concept of problem-oriented object memory in lieu of other memory sharing paradigms. Section 3 and 4 describes how this is supported and implemented in Penumbra. In section 5 we pause to discuss the notion of proxies and section 6 provides a couple of detailed examples. Section 7 discusses related work.

2 Consistency in distributed environments

The notion of consistency is the focal point around which all communication paradigms must revolve. Ideally, distributed applications are provided with an illusion of a hardware-implemented global memory giving the same strong consistency guarantees as a uniprocessor. The two dominating approaches are page-based distributed shared memory (DSM) and object-based systems.

2.1 Distributed Shared Memory

Page-based distributed shared memory, such as Ivy [18], provides a notion of a shared and consistent virtual address space on top of which one might have higher-level programming models unaware of distribution. This level of transparency comes at a cost of

- False sharing. Coherence is at the granularity of a page and data that happens to be collocated at the same page cannot be accessed independently. Furthermore, data that lie on a page boundary will typically result in multiple page faults.

- Expensive processor interrupts if the model is implemented on top of message passing without any hardware cache-coherence support.
- Inability to take advantage of application specific knowledge about access patterns of data (although some DSM systems, e.g. Munin [6], does take steps to remedy this).

2.2 Distributed Object-Based Programming Systems

A different approach to shared memory coherence is the one employed in the increasingly popular DOBPSs. These systems exhibit the following characteristics:

- The unit of consistency, distribution, communication, and migration if supported, is the *object* as opposed to, for example, processes in RPC systems, or memory pages in DSM systems. Since objects are inherently problem-oriented entities, it is more natural to express location properties in this model than in DSM systems.
- The illusion of a global address space is upheld typically through the use of proxy objects. Local and remote objects are generally treated in a uniform manner; object references can for example be passed as arguments to remote methods.
- The DSM memory access primitives are read and write of individual memory cells. The process of fetching a memory page at page fault time is called *data shipping*. In contrast, the basic means of communication in an object-based model is invocation of methods in an objects interface, i.e. *function shipping*. The equivalent of data shipping is supported through object migration.
- The fundamental parameter passing semantics is call-by-reference (also called call-by-proxy). Consistency is guaranteed as there is always only one version of an object. DOBPSs often provide mechanisms for controlling object placement for efficiency.

False sharing can also occur in an object-based system if objects are too coarse grained and different nodes access disjunct subsets of an object. This seems to us like the result of a bad design—the analysis phase should recognize such constraints and reflect them in the decomposition of functionality into objects.

It is interesting to note that naive application of either the DSM or DOBPS algorithms doesn't work well with hot data. A DSM approach would cause massive thrashing as all nodes would attempt to access the same pages at the same time, and the huge number of messages generated by a DOBPS system and the delay associated with each one would be equally prohibitive.

A different approach is obviously needed.

2.3 Problem-Oriented Object Memory

POOM is our novel model of shared memory. It combines the advantages of object-based approaches with a framework for expressing application-specific semantic knowledge about exactly what consistency means for a particular class.

The assumption is that distributed applications with a need for very high performance often doesn't need absolute consistency of shared data, but can exploit semantic knowledge of the problem domain to relax consistency [9,21].

We can identify the following types of acceptable inconsistencies (some of them identified in [9, 21]):

Tolerable. The application may simply not be very sensitive to inconsistencies in some data.

Imperceivable. The user of a shared datum will not be able to tell that the datum is inconsistent but the application might yield a suboptimal solution to its problem i.e. it degrades gracefully.

Detectable and correctable. The application will detect data inconsistency when attempt is made to use it, and is subsequently able to correct it. This requires application cooperation but may be hidden from the programmer.

Reconcilable. The synchronization required to bring shared data into a consistent state is

embedded in the access function. Unlike in the previous point no client cooperation, implicit or explicit, is required.

In the POOM model the unit of consistency is still the object, but objects are replicated for quick access and the programmer annotates method implementations of such objects with information about the type of integrity required for instances. Given an invocation on a cached or replicated version of an object, the programmer is given control over whether to execute the method locally or whether to pass it on to the real object.

Object replicas are thus allowed to slide into states which are neither consistent with the master object or with other replicas. To handle the situation where the result of queries on the master object depends on the state of replicas, Penumbra supports the notion of *collator*[2] objects. A collator object maps the state of a number of replicas or a number of replies from replicas to a single coherent object state or method result.

These ideas can be used to implement problem-specific relaxed object consistency.

An object-based model is particularly appealing to us for the following two reasons (apart from the ones mentioned above and other ones like encapsulation, inheritance, modeling, etc.):

- Network management standards are typically formulated in an object-oriented framework [23], thus making the deployment of object-based technology the natural choice.
- Resources within a distributed system can conveniently be identified and accessed through a handle to a corresponding object. The *object handle* uniquely identifies a fine-grained object within a system.

We have prototyped this model in a DOBPS called Penumbra. We found that in this framework we were able to express problem-specific object consistency protocols with a minimum of effort and with a minimum of impact on the system as a whole.

2. *Collation*: 1a: to compare critically 1b: to collect, compare carefully in order to verify, and often integrate or arrange in order (Webster's dictionary).

3 Distributed programming with Penumbra

This section describes the programming model and abstractions provided by Penumbra for enabling simple construction of distributed applications.

The programming model is that of distributed, multi-threaded objects communicating via RPC. As a result of a remote object invocation a new thread is spawned to execute the method at the current location of the object. Objects are completely symmetric, and may act both as originators and targets of RPC calls. Furthermore, object references may be passed around in the system, unconstrained by distribution. Objects can migrate between processes (modelled as Contexts in Penumbra) in the system and invocations are transparently forwarded to the objects current location.

The ability of objects to be remotely accessible, cacheable, etc. is provided in the form of a C++ class hierarchy, the idea being that shared objects inherit from the Penumbra classes corresponding to desired properties for that class. The set of abstract superclasses can be combined using multiple inheritance.

Figure 1 shows the inheritance hierarchy. The shaded classes are abstract and are derived from by applications. The other classes implement a set of standard services which represent functionality needed in most non-trivial applications. We describe each class in turn.

3.1 The abstract classes

Applications create new classes by deriving from the Penumbra base classes described in the following.

3.1.1 Distributable

Class Distributable is the basis of distribution of C++ objects. Instances of Distributable or subclasses are the units of distribution, migration and caching.

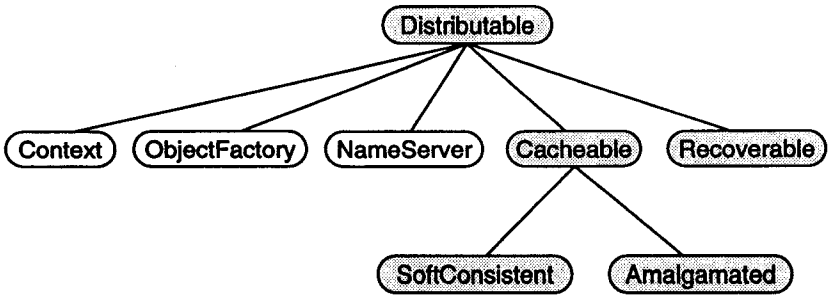

Figure 1. The Penumbra inheritance hierarchy

At the language level the use of references to remote Distributable objects is syntactically, and modulo network and processor failures also semantically, indistinguishable from ordinary use of C++ pointers. The implementation employs the well-known concept of proxies to implement this logical model: a remote object is represented locally as a shallow *proxy* object. The represented "real" object is called the proxies *principal* or *master* object [22]. The remote object is accessed only through ordinary local references to the proxy object which hides details of remote communication like argument serialization and network communication.

A Context can obtain a reference to a Distributable object in three ways:

1. It can instantiate one itself. There is nothing special about instantiation of Distributable objects compared to other C++ objects.
2. References to Distributable objects can be passed between Contexts as arguments or results of remote method invocations.
3. It can look up a name in a NameServer. If an object has been registered with that name, a reference to that object is returned.

Obtaining references to and accessing remote objects is thus syntactically identical to ordinary C++.

Class Distributable implements the `migrate` method which given a reference to an active Context, attempts to migrate the object to the corresponding process.

Low-level consistency of proxies and principals during object migration is based on object locking. Since multiple threads can execute within a single object simultaneously, Penumbra must protect threads against the situation where an invocation's controlling object is suddenly migrated by another thread. A migration request is thus carried out holding an exclusive lock for the duration of the migration while ordinary invocations are executed holding a shared lock. Of course applications can still interfere at a semantic level and will thus typically need additional concurrency control.

3.1.2 Cacheable

The Cacheable class provides objects with the ability to be cached through the `cache_object` method. In our implementation the principal object will keep track of the whereabouts of cached versions of itself. This information is needed in the subclasses implementing particular cache coherency mechanisms. The Cacheable class itself thus doesn't come with any kind of consistency guarantees but provides appropriate hooks for subclasses.

The effect of having *immutable* objects can be achieved by inheriting directly from Cacheable. Immutable objects can freely be replicated without worrying about consistency constraints.

3.1.3 SoftConsistent

Class SoftConsistent provides a traditional cache coherency mechanism. Methods which don't alter the cached object ("read" operations) are carried out locally and can thus proceed in parallel with local reads elsewhere in the system. Serialization of "write" operations is guaranteed through the use of either a *write-invalidate* or a *write-update* protocol.

The write-invalidate protocol, for example, works by letting cached objects pass on write invocations to their principal object. The principal object invalidates all cached versions of itself (in parallel), and executes the body of the method holding an exclusive lock on itself. Cache invalidation uses the information inherited from Cacheable to invoke the `invalidate` method on each cache.

Read methods invoked on a cached object checks whether the cache is valid and either executes the operation immediately and locally, or gets the new state of the object from the master and then executes locally.

The delay for executing write operations with this protocol is simply the round-trip-time for RPCs (assuming no lock contention), and the number of messages required equals the number of valid cached versions of the principal in existence at the time of the update.

Obviously the scheme must be robust towards changes in the membership of the "cached-version-of-object" predicate. Again we rely on object locking in the prototype.

The cache coherence of SoftConsistent objects can be relaxed. We provide *fuzzy* (or *chaotic*) versions of the read and write methods, which will always execute locally without any external synchronization. This allows the use of out-of-date values when application semantics can tolerate it.

Actually, fuzzy invocations are special cases of *conditional* method execution. In general, the decision about where to execute methods can be made at run-time based on the evaluation of a boolean expression. This expression is part of the method implementation and for fuzzy invocations it evaluates to false. The expression tested for in a conditional method can involve replica state as we shall see in section 6 below.

The cache can be brought up-to-date at any time by reexecuting `cache_object`.

3.1.4 Amalgamated

When there is high contention for shared data, strictly coherent caching only works well when the number of reads is (much) larger than the number of writes. Fuzzy access to caches can alleviate this problem in some cases, but assumes that the information is usable independently of other caches.

The Amalgamated class supports the process of transforming inconsistent caches to be consistent or rather to *appear* as being consistent to observers. It allows flexibility in choosing the location at which operations on replicas (cached Amalgamated objects) are carried out, and allows replicas to become inconsistent, as long as a coherent object state can be constructed from the total set of replica states.

This is useful when data is updated much more frequently than it is read. An example of this is a replicated and monotonically increasing counter which is updated on the order of hundreds or even thousands times per second but which is read only infrequently by management applications [19]. A write ratio in these orders would exclude strict consistency among replicas in our application.

One way of dealing with objects exhibiting such characteristics is to carry out updates directly on replicas without synchronizing with the principal object or with other replicated versions of the same object, and to deal with synchronization through *value amalgamation* at the time the object is read. Value amalgamation is the process of amalgamating the state of all replicas of an object to a single meaningful value.

Of course not all objects are of a nature that allows a single meaningful value to be synthesized from multiple more or less independent values (though in our application it helps that managed objects are often quite simple). Value amalgamation for such types may consist simply of making a list of individual values available to clients.

The Amalgamated class lets the user decide on a per-method basis where to execute methods and how to. The counter type may, for example, be implemented by carrying out the frequently occurring increments locally on replica versions of the counter while read requests are forwarded to the principal object. The principal will in turn retrieve the values of each of its replicated versions, and return the result of adding them together. This process is illustrated in figure 2

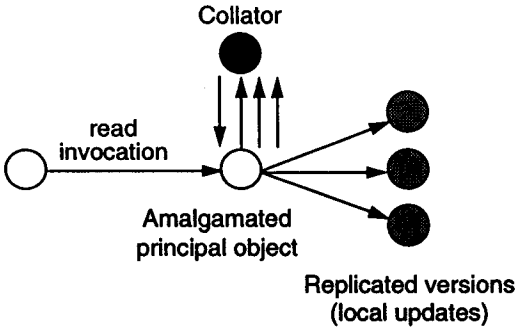

Figure 2. Amalgamated and Collator

where the Amalgamated object forwards an invocation to its replicas and delegates out to a Collator object the responsibility of mapping the three replies thus obtained on to a single reply to be returned to the client as *the* result.

This scheme is very much analogous to the logical model of multicast communication in ANSA [20]. In this analogy the master object and replicas corresponds to the group representative and group members respectively, but invocation forwarding need not be implemented using multicast protocols.

The decision about whether or not to execute a method locally within a replicated object or forward it to the principal depends on the semantics of that particular operation. We provide the programmer with macros for passing this information on to Penumbra. A more elegant solution would be to make this information part of an interface definition.

Class Amalgamated implements a method `multicast`, which (in parallel) invokes a method on each of its "cached" copies. Associated with each multicast is a collator object. The programmer specifies which collator class to use for particular methods as part of the implementation. The Collator object performs the task of producing a single result from the *n* that are received. It can discontinue the multicast at any time if it decides that enough information is available to construct the result. This sort of preemption can be exploited, for example, to trade precision for latency in the retrieval of statistical information. Replies that arrive late are silently discarded. As part of the multicast Penumbra creates a new instance of the specified subclass of Collator, feeds each multicast reply to its `collate` method as they are received, and blocks the original invocation (the read in figure 2) until the collator object comes up with a result or times out.

An advantage of this scheme is that it is possible to write generic collator classes which perform typical actions, such as return the first reply received (the default) or a list of all replies. Users with specific needs define custom collation strategies by writing new classes inheriting from Collator. The AddCollator used in the Counter example might be defined as in Figure 3.

Replica replies are passed to the `collate` method in a generic RpcBuffer since the same method prototype must allow different types of values in different subclasses.

We see the notion of collation filters as supporting the construction of "democratic memory" as defined in [9]. Democratic memory deals with fetch requests by reading the values of its replicas and returning the result of some computation on the memory values thus received.

3.1.5 Recoverable

The Recoverable class provides objects with the ability to persist on secondary storage independent of the process creating the object. The implementation exploits the marshalling code already present for distribution purposes.

3.2 Standard services

We have identified the Context, ObjectFactory, and NameServer classes as providing commonly used services. These classes all inherit from Distributable and instances are thus remotely accessible.

3.2.1 Context

The Context class is an abstraction of processes. In the spirit of network management standards, a process is considered a resource, and is subsequently modelled as an object, towards which invocations (remote and local) can be issued, thus controlling various aspects of the corresponding process.

```
class AddCollator: public Collator
{
 public:
   ...
   virtual void collate(RpcBuffer &reply_buff); // invoked for each reply
 private:
   int accumulated;                            // sum of replica values
};

void AddCollator::collate(RpcBuffer &replyBuff) // replyBuff contains reply
{
   int incr;

   reply_buff >> incr;                         // unpack the reply
   accumulated += incr;                        // add to interim result
   if (repliesReceived == numCaches)           // if this was last reply
   {
      resultBuff << accumulated;               // pack return value
      return_result(resultBuff);               // discontinue multicast
   }
}
```

Figure 3. An example Collator class definition

Apart from encapsulating per-process data structures, the Context implements methods `activate` and `kill`. An application can explicitly instantiate new Contexts. These will start in the *passive* state and can be activated on a (possibly remote) node through the activate method. A process will be started on the node determined by an argument (assuming that the environment permits it) and the Context object will change to the *active* state and will become a proxy for the master object in the new process.

The kill method terminates the process corresponding to that Context.

A typical use of Contexts is to designate the target of object relocations.

3.2.2 ObjectFactory

The ObjectFactory class supports runtime creation of instances of distributable objects through the `create_object` method which all applications must reimplement. It returns a handle to a newly created object of the specified type or `NULL` if it fails. This can be used directly by applications but is heavily used internally. If the ObjectFactory doesn't have a priori (i.e. static) knowledge of the class it is being asked to create an instance of, it will attempt to dynamically load the code for it from a shared library.

3.2.3 NameServer

The name server in our (and similar) systems provides the indirection in binding object references to process and object addresses that is necessary for writing location transparent applications. It is a standard component in many high-level distributed object-based system.

All Penumbra processes are born with a reference to the system-wide name server and will connect to it as part of their start-up sequence. Multiple NameServers can, however, exist in a system, and a process can reference and access more than one of these. Likewise, NameServer references can themselves be inserted into other NameServers thus providing a hierarchy, or federation, of NameServers.

4 Implementation

As already mentioned, remote objects are locally represented as proxies and are locally referenced as ordinary C++ objects using their virtual memory address. No special precautions are taken to ensure identical layout of virtual memory in communicating processes. (This would imply a process coupling which we didn't find suitable in our environment. Amber, which assumes a tight coupling between communicating processes, does this [8]). Instead object references are mapped to new virtual memory values when they are passed between processes.

For this purpose each Distributable object is assigned a unique object identifier (UID) and each Penumbra process maintains a mapping between UIDs and pointers to local principals/proxies. Instead of passing a pointer to some Distributable object the sending process passes the corresponding UID. The receiving process is then able either to map this UID to an already existing local principal/proxy or to instantiate a new proxy and update the UID-to-pointer mapping.

When an object is migrated it leaves behind a proxy, which acts as a forwarding address. If an object is repeatedly migrated, chains of remote references can arise. Old remote references are thus left out-of-date and are updated only when dereferenced or when more up-to-date information appears, and always transparent to users. Forwarding chains can thus be seen as a problem-oriented mechanism: instead of keeping references consistent the system infra-structure support the detection of stale data on use with subsequent correction.

Internally, Penumbra needs a way of identifying and communicating types as if they were first class values. For simplicity we identify types as text strings in the prototype. All Distributables respond to the virtual `get_type` method with their type name. There is no check that processes are compiled with identical versions of classes.

Also for simplicity, proxies are actually instantiations of the same class as the corresponding principal object, only marked as being a proxy. This means that standard compilers can more readily be applied and that object migration isn't complicated by the need to relocate local pointers from the existing proxy to the (locally) new principal.

Multithreading in Penumbra is based on DCE threads which complies to POSIX threads draft 3.4. Concurrency within a process is controlled using the features of DCE threads, namely mutexes and condition variables.

5 Transparency issues

The notion of proxy objects can be seen as the object-oriented analogue to RPC stubs in that they allow location transparent access to objects. They can, however, also be seen as supporting other types of transparencies such as:

Caching transparency. The concealment of the fact that cached objects are any different than ordinary ones from a particular user and the insurance of cache consistency using whatever protocols are appropriate.

Replication transparency. The concealment of object replication from a particular user—again any consistency constraints can be transparently enforced by the system.

Group transparency. When an object-based system is extended to include some notion of group communication, a group should be represented by an object like all other entities. Clients interface to the group via this *group object*, and invoke operations on the group by invoking operations on this object [16]. Group transparency entails hiding of invocation distribution, group membership changes, multiplicity of replies, etc.

The commonality between these examples and the common use of proxies is that they all employ local objects to represent remote resources and to hide complex distributed algorithms, and also that they do this by stretching the notion of unique object identity: an ordinary proxy has the same UID as its principal (we think of it as the same object). Group objects resemble ordinary proxies in that they pass invocations on to other objects but unlike ordinary proxies they don't directly correspond to something

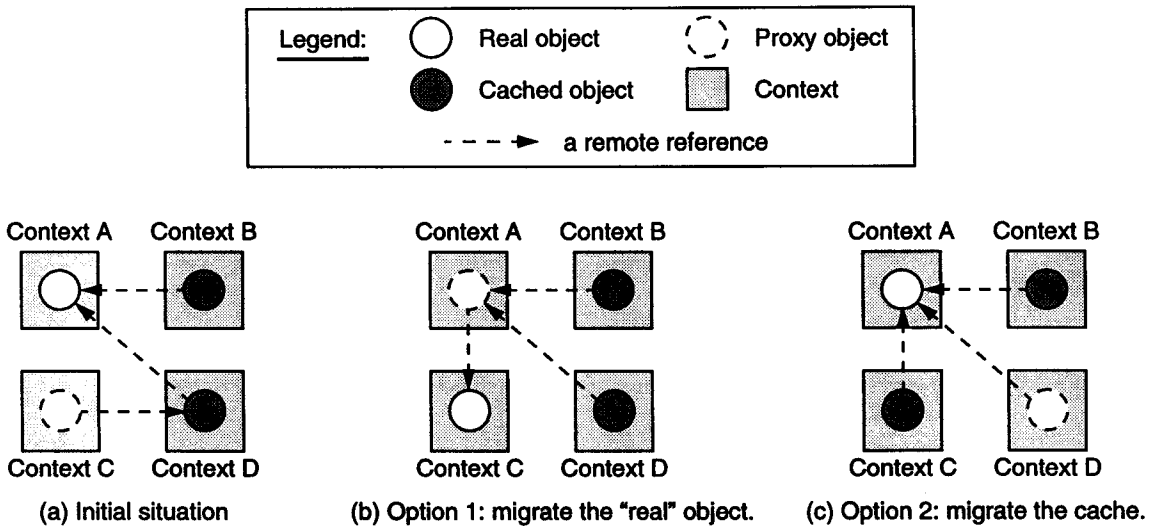

Figure 4. Migration of cached objects

more "real" than themselves. Replicated and cached objects can be seen as variations over the same theme.

Transparencies correspond to different levels of abstraction. Since different users have different perspectives on objects, they will need different abstractions and it is thus important that transparencies are *selective*, i.e. that mechanisms exist for escaping them. The location of an object may for example be very important for performance or fault tolerance reasons. Or a client of an object group might really want to retrieve a result from each group member.

A transparency is the concealment of some property from the user. In Penumbra transparencies come with classes such as Cacheable. Other transparencies can be introduced by refinement through single inheritance (Cacheable inherits from Distributable) and combined by multiple inheritance.

An important assumption is that transparencies can be made orthogonal in the sense that the result of combining them will adhere to the individual consistency requirements. One can, for example, freely pass references to cached objects around since these are also Distributables. Or as another example consider what happens when one migrates a cached version of an object. There are several possibilities as illustrated in figure 4.

In Figure 4(a) site C has a proxy for a cached version of an object residing at site D. The principal is located at site A. Figures 4(b) and 4(c) depicts two possible answers to the question about what happens when someone invokes `migrate` on site C's proxy giving it a reference to Context C as argument. The proxy passes the invocation on to the next element in the chain, i.e. to the cached object at site D, which in turn has a choice between

1. passing the migration request on to the principal; this results in the migration of the principal object to site C, or
2. executing the request itself, that is migrating the cache instead of the real object to site C.

Option 1, which is what we have implemented in Penumbra, seems like the natural choice of semantics, but option 2 would be equally consistent. Other possible "feature interactions" may have a less obvious resolution.

It seems that the notion of proxies in an object-oriented environment makes an ideal transparency mechanism. They provide the indirection which enables the run-time system to perform certain actions depending on the nature of the object.

6 Examples

We will now turn to consider two concrete examples of how problem-oriented protocols can greatly decrease communication by relaxing consistency.

They exemplify the notions of tolerable and correctable inconsistencies respectively. An example of reconcilable inconsistency was given in section 3.1.4.

6.1 The gauge threshold counter

The example is a threshold counter as it appears for example in the context of OSI network management. A threshold counter has associated with it an increment and a decrement operation, and a number of threshold pairs. Whenever the high-water mark of a threshold pair is crossed in upwards direction, a *ThresholdExceeded* notification is generated and whenever a low-water mark is crossed in downwards direction a *ThresholdAbated* notification is generated[3]. Figure 5 shows a gauge with only one associated threshold pair. Thresholds are specified in pairs so as to avoid oscillations around one value to cause huge numbers of notifications to be generated.

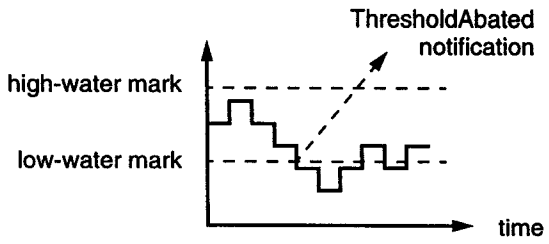

Figure 5. The threshold counter

When users of a gauge are functionally replicated, as is the case in our application, the straightforward approach is to have a centralized gauge object and do increments and decrements as ordinary remote operations on it. However, we can achieve huge savings in remote communication by weakening the precision of threshold detection.

3. Notifications are subsequently processed and forwarded to any interested parties.

Instead of performing each increment and decrement as remote operations on the principal object, we increment and decrement in larger units. We refer to this as the *N-indeterminate integrating gauge*, see Figure 6. The N determines the application dependent

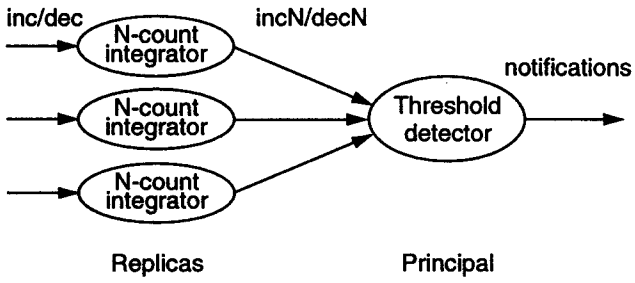

Figure 6. The N-indeterminate integrating gauge

trigger precision: inc's and dec's are done on the master object in units of N/2. The case where N is less than or equal to 2 degenerates to the default action of forwarding *each* update to the master. The point is that delaying threshold detection often isn't critical in practice [19], i.e. this is an example of a tolerable inconsistency.

Figure 7 shows the result of applying the integrating gauge to an application. The two plots show how one of the N-count integrators from Figure 6 react to a series of inc's and dec's for different values of N. Increments and decrements are generated as the result of two independent poisson processes intended to simulate real traffic. Plot 1 where N equals 6 reduces 150 updates to 14 going remote, while a value of N = 10 reduces the number to only 4, as shown in plot 2. These numbers correspond to a reduction in network traffic by factors of 10.7 and 37.5 respectively.

The integrating gauge example was implemented using the conditional invocation forwarding feature of Cacheable.

Network management standards define a multitude of objects resembling the threshold counter. One variation is where the master object itself takes the initiative to update its value. In this scheme, the master is an active object, conceptually or physically executing in parallel with other objects, and it

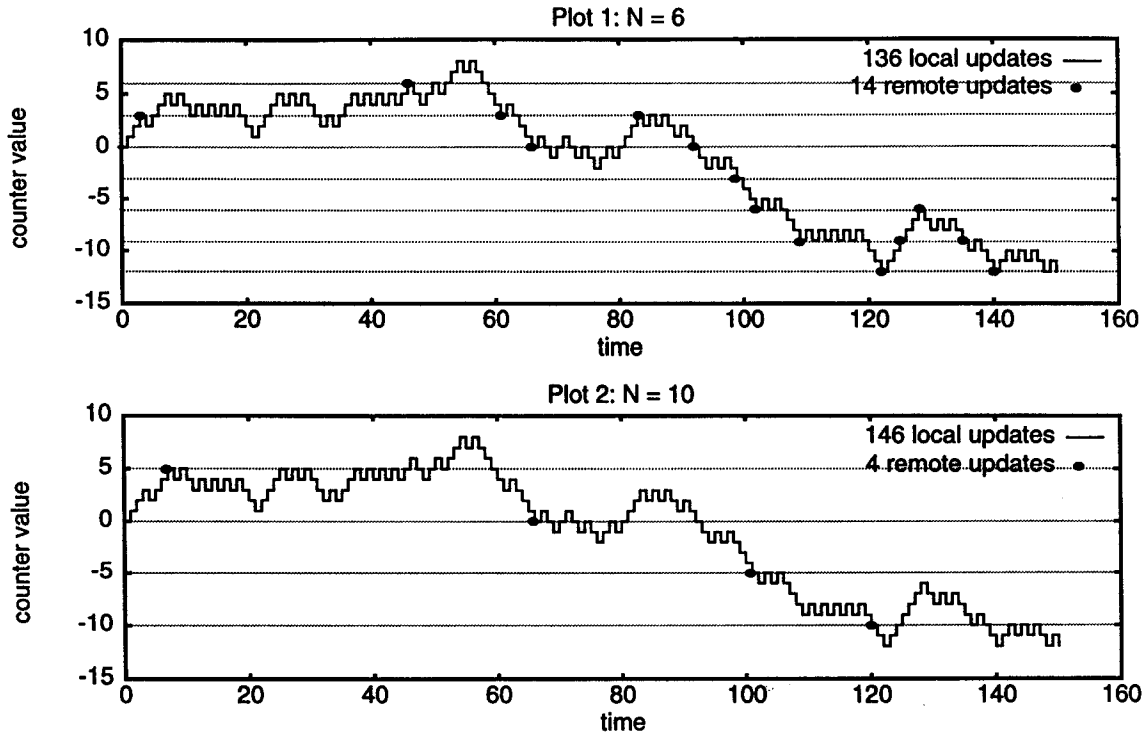

Figure 7. The integrating counter on simulated traffic

periodically poll replicas using the `multicast` method, thus amalgamating their state to form its own new state.

The idea of only maintaining *approximately* accurate information applies especially well to statistical information, such as the average number of requests during some time-interval or the rejection rate for connection requests.

6.2 Mobile station location information

In section 4 it was noted that proxy forwarding chains can be argued to be an example of a problem-oriented mechanism. Out-of-date references are allowed because of the increase in performance due to less synchronization at object-migration time. When an application attempts to access an object using a stale reference it is detected and the object reference transparently updated.

This example has an almost equivalent telecoms formulation. In mobile communication networks, information about the whereabouts of a mobile subscriber is held in *location registers*. The smallest notion of subscriber location is the *cell* while a number of cells are grouped together to form a *location area*.

There's a trade-off between exactness in location registration and overhead in locating a user on incoming calls. Typically the network has approximate knowledge of subscriber location—it knows the location area but identifies the exact cell by paging all cells controlled by that location area, i.e. through a form of broadcast [14].

This solution strikes a better balance between overhead of updating location information and paging than does the two extreme solutions of maintaining exact location information and paging the entire world on incoming calls.

7 Related work

Penumbra is a DOBPS in the tradition of programming languages like Amber [8], Emerald [4,15], DOWL [1], Distributed Smalltalk [e.g. 10] and BETA [5]. These systems all implement strict consistency,

essentially by only ever having *one* version of an object. Applications can optimize remote communication by specifying object *attachment* [15], i.e. that some objects must migrate together, and by specifying *call-by-move* and *call-by-visit* parameter passing semantics, i.e. that the argument migrates with the invocation either permanently or for the duration of the call. These systems have no support for object replication.

Cheritons work on *problem-oriented shared memory* is formulated in the framework of low-level memory accesses, i.e. reads and writes of memory locations [9]. He identifies a number of areas where problem-oriented approaches to relaxing strict cache coherency can yield substantial performance improvements. In [21] the use of partially ordered multicast algorithms is used to support relaxed consistencies. Our work differs mainly in integrating a problem-oriented approach with an object-oriented distributed programming model. Our claim is that expressing problem-specific needs at the semantic level of objects is both more natural and allows better localization of protocol details.

In the Orca language data sharing is provided through automatic replication of *data objects* [3]. Objects are kept consistent for example by using atomic update protocols. It is thus best suited for data that is frequently read and infrequently written. No support for explicit object migration is provided.

Several proposals have been given for the use of relaxed memory models in non-uniform memory access multiprocessor architectures, including *weak ordering* [2] and *release consistency* [12].

These approaches are also applicable to more loosely coupled machines not implementing cache coherence in hardware, but then suffers from processor interrupts and typically also higher memory latency. Systems implementing a relaxed consistency model guarantee that applications will observe sequential consistency (as defined in [17]) *provided* that they make synchronization operations visible to the system. The provision of this semantic information allows the hardware or runtime system to provide strong ordering guarantees with a reduction in the overhead and number of messages exchanged.

Munin is an example of a page-based DSM system which implements release consistency (together with multiple consistency protocols) [6]. In Munin, applications are explicitly annotated with information about where synchronization needs to take place, and is thus able to achieve performance quite close to that of a message passing solution on certain problems.

Versioned distributed object memory is a variation of the weak memory model in which i) objects are immutable, modifying an object results in the creation of a new *version* of that object, and ii) on each access the programmer selects which version of an object on which to operate [11]. This form of weak consistency is thus mostly useful if such ordering information is already implicitly or explicitly available in the application. This is claimed often to be the case in scientific numeric applications.

In a sense the problem-oriented approach goes further in its use of semantic information than the weak memory models—the goal isn't to arrive at sequential consistency but merely at an application-specific notion of consistency. We can therefore achieve very good performance improvements for a restricted set of applications.

The difference between a hardware based cache coherence protocol and the notion of problem-oriented object memory can also be highlighted by noting that they could be combined in a shared-memory multiprocessor to achieve higher performance than any of the two approaches in isolation.

8 Conclusions

We have demonstrated how a problem-oriented approach to memory consistency can successfully be combined with an object-based distribution toolkit. In the problem-oriented object memory model, objects are replicated for performance and consistency of shared objects is relaxed depending on semantic information.

We have identified and prototyped support for different types of object consistency based on:

- Ordinary proxies and remote invocations (i.e. function shipping).

- Immutable objects.
- Caching and strict cache consistency.
- Fuzzy reading and/or writing of replicated objects.
- Value amalgamated (democratic) memory.
- "Almost" consistent objects. Exemplified by the ThresholdCounter.

It is an important benefit of problem-oriented object memory that it allows cached and replicated objects to be accessed in a manner that is uniform with ordinary object access. Also, the use of objects allows application-specific update protocols to be isolated from clients of the object, thus localizing and minimizing such information.

Though any application of POOM will inherently be application dependent it is our belief that a problem-oriented approach is applicable to a large class of problems, not only in a network management context, and that the object-oriented programming paradigm provides the right framework for it.

Acknowledgments

We would like to thank Steve Caughey for making the Shadows toolkit available to us for experimentation and for tirelessly answering our questions. Also, much appreciated detailed comments were provided by Søren Brandt, Claus Pedersen, Andrew Thomas, Rene Wenzel, and the anonymous referees.

References

[1] Bruno Achauer. The DOWL distributed object-oriented language. *Communications of the ACM*, 36(9):48–55, September 1993.

[2] Sarota V. Adve and Mark D. Hill. Weak ordering—a new definition. In *17th Annual International Symposium on Computer Architecture*, pages 15–26, 1990.

[3] Henri E. Bal and Andrew S. Tanenbaum. Distributed Programming with Shared Data. In *Proceedings of the International Conference on Computer Languages*, pages 82–91. IEEE, October 1988.

[4] Andrew Black, Norman Hutchinson, Eric Jul, Henry Levy, and Larry Carter. Distribution and abstract data types in Emerald. *IEEE Transactions on Software Engineering*, 13(1), January 1987.

[5] Søren Brandt and Ole Lehrmann Madsen. Object-oriented distributed programming in BETA. In Rashid Guerraoui, editor, *Object-Based Distributed Programming*, number 791 in Lecture Notes in Computer Science, Berlin, 1993. Springer Verlag.

[6] John B. Carter, John K. Bennet, and Willy Zwaenepoel. Implementation and performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 152–164, October 1991.

[7] S. J. Caughey, G. D. Parrington, and S. K. Shrivastava. SHADOWS—a flexible support system for objects in distributed systems. In *Proc. of 3rd IEEE Intl. Workshop on Object Orientation in Operating Systems*, pages 73–82, Asheville, North Carolina, December 1993.

[8] J. Chase, F. Amador, E. Lazowska, H. Levy, and R. Littlefield. The Amber system: Parallel programming on a network of multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating System Principles*. ACM, New York, 1989.

[9] David R. Cheriton. Problem-oriented shared memory: A decentralized approach to distributed system design. In *Proceedings of the 6th International Conference on Distributed Computing Systems*, pages 190–197, 1986.

[10] Dominique Decouchant. Design of a distributed object manager for the Smalltalk-80 system. In *Proc. ACM Conf. Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, Portland, OR, 1986. ACM.

[11] Michael J. Feeley and Henry M. Levy. Distributed shared memory with versioned objects. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, October 1992.

[12] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event

ordering in scalable shared-memory multiprocessors. In *17th Annual International Symposium on Computer Architecture*, pages 15–26, 1990.

[13] ISO 10165-2. Information processing systems—open systems interconnection—structure of management information—part 2: Definition of management information.

[14] Bijan Jabbari. Intelligent network concepts in mobile communications. *IEEE Communications Magazine*, 1(2):64–69, February 1992.

[15] E. Jul, H. Levy, H. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[16] Anders Kristensen. Object-oriented group communication in BETA. Master's thesis, Computer Science Department, Aarhus University, June 1994.

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

[18] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. In *Proceedings of the 5th ACM Symposium on Principles of Distributed Computing*, pages 229–239, August 1986.

[19] Colin Low and Anders Kristensen. SingleView—high performance single systems view for telecoms applications. Technical report, HP Laboratories, Bristol, February 1995. Forthcoming.

[20] Michael H. Olsen, Ed Oskiewicz, and John P. Warne. A model for interface groups. In *Tenth symposium on Reliable Distributed Systems*, Pisa, Italy, October 1991.

[21] K. Ravindran and Samuel T. Chanson. Relaxed consistency of shared state in distributed servers. In Thomas L. Casavant and Mukesh Singhal, editors, *Readings in Distributed Computing Systems*, pages 133–151. IEEE Computer Society Press, 1994.

[22] Marc Shapiro. Structure and encapsulation in distributed systems: The proxy principle. In *Proceedings of the 6th International Conference on Distributed Computing Systems*, Cambridge, MA, May 1986.

[23] William Stallings. *SNMP, SNMPv2, and CMIP*. Addison-Wesley, 1993.