MIKE: A Distributed object-oriented programming platform on top of the Mach micro-kernel

Miguel Castro, Nuno Neves, Pedro Trancuso, and Pedro Sousa
INESC, Portugal

MIKE
A distributed object-oriented programming platform on top of the Mach micro-kernel

Miguel Castro, Nuno Neves, Pedro Trancoso and Pedro Sousa

email: (miguel,nuno,pedro,pms)@sabrina.inesc.pt

INESC - R. Alves Redol n°9 1000 Lisboa PORTUGAL

Abstract

This paper describes the architecture and implementation of MIKE - a version of the IK distributed persistent object-oriented programming platform built on top of the Mach microkernel.

MIKE's primary goal is to offer a single object-oriented programming paradigm for writing distributed applications. In MIKE an application programmer can use C++ almost as he would in a non-distributed system.

The platform supports fine grained objects which can be invoked in a location transparent way and whose references can be exchanged freely as invocation parameters. These objects are potentially persistent. MIKE supports the abstraction of one-level store, persistent objects are transparently loaded on demand when first invoked and saved to disk when the application terminates. Class objects are special persistent objects which are dynamically linked when needed. The platform also offers distributed garbage collection of non-persistent objects.

This paper discusses how MIKE makes use of Mach's features to offer the functionality described above and the techniques used to achieve good performance. MIKE is compared with the UNIX *versions of IK to evaluate the benefits of using Mach abstractions.*

1 Introduction

Writing distributed applications is still a difficult task due to the lack of adequate support. In a traditional environment a programmer uses a general purpose programming language to program most of the objects and an Interface Definition Language (IDL) to generate stubs to access remote ones. Generally, the IDL concepts do not match those of the programming language – remote objects are named and accessed in a different way, parameter passing is different and inheritance does not exist or has different rules. All these problems tend to separate the application into static sets of co-located objects that communicate with remote ones using Remote Procedure Call (RPC) based client-server interfaces.

MIKE, Mach IK Environment, is a redesigned version of the INESC Kernel (IK [9, 13]) that takes advantage of Mach's [1] features. IK's primary goal is to offer a single object-oriented programming paradigm for writing distributed applications. In the current versions, applications are written in EC++, a language with the same syntax as C++ but with some restrictions and semantic extensions [11]. It supports fine grained objects which can be invoked in a location transparent way across the distributed system and whose references can be freely exchanged as invocation parameters. An application can be designed as a set of fine grained communicating objects whose location in the system can be dynamically set at run-time.

Traditionally, a programmer uses files to persistently store objects and has to handle the conversion of all context dependent data (e.g. pointers) when saving or retrieving objects. IK solves these problems by offering the abstraction of a one-level store – all objects are potentially persistent and persistent objects are transparently loaded when invoked and saved to disk when the application terminates. Class objects are special persistent objects which are dynamically linked to running applications when needed. A new version of a class object can replace an old one at run-time, provided it preserves the old interface and semantics.

IK's implementation uses only UNIX abstractions to achieve portability. It runs on a network of Sun workstations, PC and Bull machines running different UNIX flavors. MIKE shares its goals and model with IK but bases its implementation on Mach abstractions. The MIKE prototype is running on the Mach 3.0 micro-kernel and the BSD single server.

There are several demonstration applications running, from which we outline a distributed ray-tracer. This ray-tracer is designed as a set of MIKE objects which implement the processors farm model. These objects are distributed through several nodes to achieve true parallelism and speed-up image rendering.

2 Overview

MIKE uses an object-oriented approach, all entities are objects conforming to a simple conceptual model. This model was inherited from IK and it is fully described in [13].

2.1 Basic abstractions

The platform offers the following set of basic abstractions:

Object – An object is a passive data structure which exports a set of methods defined by its `class` (or by its class superclasses). Objects can be volatile or persistent. Persistent objects survive application termination. Objects can be invoked in a location transparent way.

Activity – An activity is an active object which represents a thread of control. Activity synchronization is based in lock and condition objects. Like all objects, activities, locks and conditions can be invoked in a location transparent way.

Context – A context is a protected address space mapping a set of objects. The context itself is represented by an object. This object has methods to create activities in its own address space. Several activities run in each context.

Node – A node is an object which represents a machine in the distributed system. Activities invoke a node object to create a context on its corresponding machine. Several contexts can coexist in the same node.

Mach offers abstractions that match these closely. A Mach port can be associated with distributed objects and send rights can be used as object references. Mach Inter-Process Communication (IPC) can be used to access remote objects in a location transparent way and the Mach interface Generator (MiG) stubs can be used to marshall and unmarshall invocation parameters. The context abstraction matches closely a Mach task – a resource container shared by several threads of execution. An activity can be mapped on a Mach *Cthread* and locks and conditions can be mapped on *mutexes* and *condition variables* from the Cthreads package. These observations guided our implementation on top of Mach.

2.2 Architecture

MIKE runs on several computing nodes connected by a network and running the Mach 3.0 NORMA kernel [1] and the BSD single server. The platform is composed of a set of node servers, storage servers (SS), name servers (NS) and contexts. This architecture is depicted in figure 1.

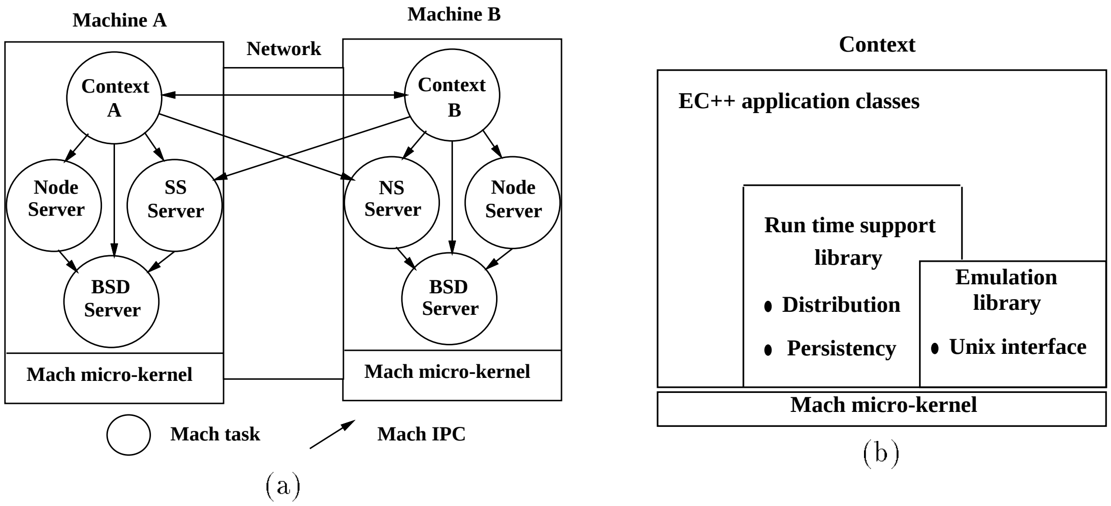

Figure 1: MIKE's architecture – possible configuration (a) and interface layering (b).

There is a node server on each machine. It boots the other MIKE servers on that machine, creates local contexts on behalf of activities on remote nodes and authenticates users.

The storage servers manage persistent objects' disk images. These servers are responsible for the generation of persistent object identifiers. They also offer a service that returns an object location given its persistent identifier.

The name servers cooperate to offer a distributed name service. This service associates names (strings) with objects. These associations are persistent.

The node, storage and name servers are multi-threaded UNIX processes. They use the UNIX functionality provided by the BSD server to access the file-system and communicate with the client contexts using Mach IPC.

EC++ application classes can use the UNIX interface or directly the micro-kernel interface. MIKE features are supported by the run time support library. Each context's process

[1] MIKE also runs in the non-NORMA kernels, but without distribution because the netmessage server does not work properly.

image is linked with this library. The run time support library offers transparent object invocation, dynamic linking, garbage collection of distributed volatile objects and cooperates with the servers to support object persistency. This functionality is implemented using mostly the micro-kernel interface.

The transparent object invocation abstraction hides the object location from the programmer. If an activity invokes a persistent object, which is in disk, the object is transparently mapped on the activity's context. Similarly, when an activity invokes an object mapped in a remote context invocation is transparently forwarded to that context.

A port is associated with each distributed object and remote invocation is based on MiG generated stubs. These stubs are encapsulated in EC++ classes which implement the mechanism described above. The MiG client stub is encapsulated in the client proxy class. This class exports the same interface as the original class. The server proxy class encapsulates the server stub. Each context has a port set where all object ports are inserted. A set of dispatcher threads wait for requests in this port set. They service remote invocation requests directed to the objects mapped in the context. The remote invocation mechanism is used to share objects between contexts.

3 Implementation

MIKE reuses part of IK's implementation, but distribution support is significantly different. MIKE also adds real multi-threading support and distributed garbage collection.

3.1 Object structure, local references and object invocation

Instance objects and their classes are represented in memory by the structure shown in figure 2. Objects are represented by Run Time Headers (RTH). The RTH hold all relevant per-object information. Local object references are pointers to these headers.

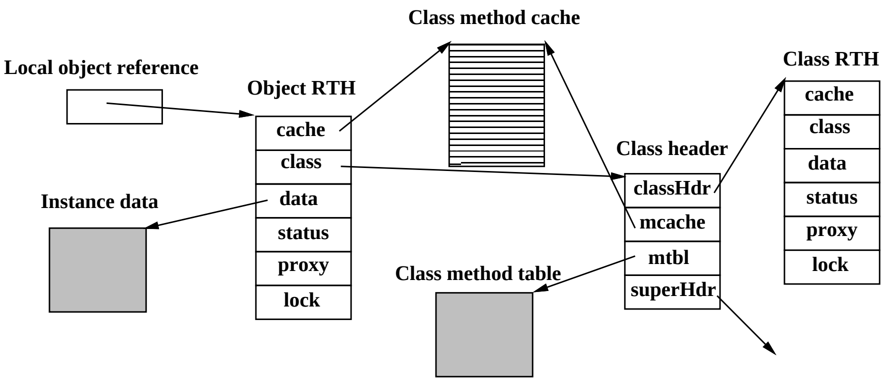

Figure 2: In memory object structure.

The `data` field in the RTH points to the object's instance data. This indirection simplifies object mobility. IK's conservative copying garbage collector [5] relies on this indirection. It also simplifies the implementation of lazy evaluation techniques like on demand dynamic linking and delayed pointer swizzling. On the other hand, accesses to the instance data are penalized.

The `status` field is used to indicate the object's type (e.g. whether the object is a class, a client proxy, a server proxy or a regular object) and its status (e.g. whether the object is mapped or unmapped).

The `proxy` field is used to store information relevant to the remote invocation process (see section 3.3).

The `lock` field stores the object lock, which is used to ensure mutual exclusion in RTH accesses. We chose to use a spinlock instead of a relinquishing mutex because the size of the first one is one fifth of the size of the second. Furthermore, our measurements show that a spinlock is more efficient than a mutex.

Method selection is performed by searching for the method in the class hierarchy, using a late-binding mechanism similar to the one used in Smalltalk and Objective-C. Each class has a method's cache which stores the entry points for the most recently invoked methods, speeding up invocation. The `cache` field of an object's RTH points to the method's cache of its class. The per-class method's cache is shared by several threads and accesses must be synchronized using the lock in the class RTH.

The detection of object faults (i.e. the invocation of unmapped objects) is achieved by trapping object invocations. This prevents direct accesses to the instance data. When an object is invoked, the invocation primitive looks for the method in the cache. Only if there is a cache miss is the object status tested to confirm whether it is a normal cache miss or an object fault. Hence, the RTH of unmapped objects always point to an empty method's cache.

The object fault handler tests the status field. If the object is persistent and not mapped anywhere, it retrieves the object from persistent store. Otherwise, if the object is a client or server proxy, the object fault handler creates an instance of the corresponding class. In either case, the object's class hierarchy is dynamically loaded and linked. Several object faults can be handled concurrently but not on the same object. Hence, the object fault is handled under the protection of the object lock.

3.2 Distributed object references

Most objects are known only inside a single context and are identified using only local references. When objects get referenced by persistent names or by other contexts they are assigned more expensive identifiers.

MIKE associates a Mach port with objects known remotely and remote contexts use send rights to reference them. This scheme has several advantages – location transparency, notifications of port destruction, port reference counting and protection by capabilities [3]; but it does not solve all problems.

3.2.1 Referencing potentially persistent distributed objects

On the one hand persistent objects survive the death of all contexts referencing them and, on the other hand, ports are volatile entities – a port is destroyed along with its associated rights when the task holding the receive right terminates. Therefore send rights are inadequate for use as distributed references to persistent objects. Persistent objects must be identified using persistent identifiers.

Since all objects are potentially persistent, using send rights as the only remote references to volatile objects is also problematic. If a volatile object gets promoted to persistent there is no easy way to transmit the new persistent identifier to all the contexts holding send rights. This problem is solved by assigning a persistent identifier, *a priori*, to a volatile object, the first time a reference to that object is exported.

MIKE's persistent identifiers are called Low Level Identifiers (LLI). They are unique and persistent, i.e. the identifier is valid until the object it refers to is explicitly deleted. The LLI is sufficient to identify the object. However MIKE uses the tuple <*oLLI, sRight*> as a distributed object reference to retain the benefits of using send rights. In this tuple *oLLI* is the object's LLI and *sRight* is a send right to the port associated with the object.

Remote invocations always try to use the send right. If it is invalid, the LLI is used to question the storage server responsible for the object and either get a new send right or map the object locally (see section 3.5.2).

3.2.2 Lazy port allocation

Associating ports with objects has all the advantages enumerated above but can consume a significant amount of kernel resources. Creating a port for every object is not feasible and it is also inefficient; because the number of objects is usually very large and most of them are short lived and never become known outside their creation context (e.g. one of our test applications, a cooperative document editor, creates 250 objects per second).

We use lazy evaluation techniques to minimize this problem – a port is only associated with a mapped object the first time a reference is exported. Furthermore, if the object is persistent and is currently not mapped, no port is associated with the object, only the LLI is sent. This significantly reduces the number of ports used.

3.2.3 "On-the-wire" reference representation

A context references a remote object using the tuple <*oLLI, sRight*>. However, to invoke the object, the LLI of the object's client proxy class is also required. Hence, a reference is transmited as the tuple <*oLLI, sRight, cpLLI*>, where *cpLLI* is the client proxy class' LLI, whenever all the information is locally available. This default policy can be changed on a per-class basis to enhance performance.

3.2.4 Exporting object references

The primitive used to export references (`XRef`) converts local object references into the tuple <*oLLI, sRight, cpLLI, pType*>. The first three components are the "on-the-wire" reference representation and *pType* is the port type. *pType* is needed because object ports are declared polymorphic on the sender's side [8]. The proxy classes call `XRef` for each exported reference.

`XRef` tests the object's status. If the object does not have an associated server proxy, `XRef` associates an LLI, a port and an instance of the corresponding server proxy class with the object. The port and the server proxy's RTH are allocated together to ensure that the port name matches the RTH's address. Collisions are rare enough for this process to be efficient and it saves a hash table and speeds up message dispatching.

The port name is stored in the `proxy` field in the object's RTH and the object's port is inserted in the context's port set. The `cache` field in the server proxy's RTH is initialized with a pointer to an empty method's cache and the object reference is stored in the `proxy` field.

Since distributed garbage collection is based in no-more-senders notifications, the send right for a locally mapped object must be generated using `mach_port_insert_right` under the protection of the object's lock. This lock synchronizes reference exporting with the no-more-senders notification handler. The value of *pType* returned in this case is `MACH_MSG_TYPE_MOVE_SEND`. On the other hand, if the local reference points to a client proxy's RTH, *pType* is returned with the value `MACH_MSG_TYPE_COPY_SEND`. Therefore, the extra send right is generated by the `mach_msg` call avoiding the extra system call.

3.2.5 Importing object references

The primitive used to import references (`IRef`) converts the tuple <*oLLI, sRight, cpLLI*> into a local reference. The proxy classes call `IRef` for each imported reference.

`IRef` starts by searching for a RTH associated with *oLLI* in a hash table. If the search fails a RTH is created and associated with the LLI. This association is registered in the hash table. Once again, the RTH's cache field is initialized with a pointer to the empty method's cache.

When *sRight* is valid, `IRef` must handle the extra user reference to the send right. If the object is mapped locally, the send right received must be destroyed to enable no-more-senders notifications. On the other hand, if *sRight* corresponds to a locally mapped client proxy the reference can be discarded lazily – a counter is incremented and the extra references are discarded when it reaches a high-water mark.

If *sRight* is valid the object it refers to is mapped somewhere. Therefore, if the object was not known yet *cpLLI* and *sRight* are saved in the `data` and `proxy` fields of the RTH and the RTH is marked as belonging to a client proxy.

3.3 Remote object invocation

In a distributed object-oriented system like MIKE, remote object invocation is a fundamental primitive. The programmer must be able to program and invoke remote objects like local ones with acceptable efficiency. If the invoking object and the invoked one are co-located, invocation proceeds locally and without interposition of any proxy object, otherwise a remote invocation mechanism is used.

The remote invocation mechanism relies on three objects:

- An instance of the client proxy class mapped in the remote invoking context. This instance holds a send right to the port associated with the remote object it represents.
- An instance of the server proxy class mapped in the context of the invoked object which represents all the contexts holding references to the object. It holds a receive right to the object port and a reference to the object.
- The object itself mapped in the invoked context. Its class knows nothing about communication and remote invocations.

A remote invocation begins when the client proxy is invoked. The client proxy calls the MiG client stub. They marshall the invocation parameters and send an IPC to the remote object port. In the remote context, the dispatcher thread finds the appropriate server proxy and invokes it. The server proxy and the MiG server stub unmarshall the invocation parameters and invoke the object. Control is returned to the client context through the same path.

3.3.1 Proxy class generation

The code for the proxy classes is automatically generated from the original class definition. This process is depicted in figure 3. All the actions are performed by a shell script, in a way transparent to the programmer.

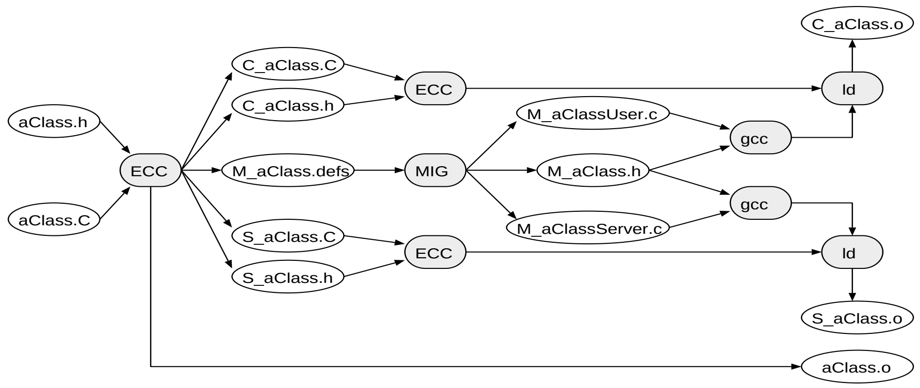

Figure 3: Proxy class generation. `ECC` is the EC++ compiler.

In the first step ECC generates the client proxy class (C_aClass.[Ch]), the server proxy class (S_aClass.[Ch]) and the MiG definition file (M_aClass.defs). In the second step the proxy classes are compiled and the MiG definition file is processed. The third step compiles the MiG output files and links them with the corresponding proxy class. In the last step (not shown in the figure) class objects are created and an LLI is associated with each class.

3.3.2 The proxy classes

To describe the proxy classes we use class *aClass* as an example:

```
struct aStruct {
   char aChar;
   int  anInt;
};

class aClass : public bClass {
public :
   virtual char aMethod(aStruct, int*, object*);
};
```

This class definition is processed as described previously.

The MiG definition file generated is as follows:

```
subsystem M_aClass 5;

userprefix call_;
serverprefix do_;

routine aClassaMethodF7aStructPiP6object
(
          port  : mach_port_t;
   in     a1_1  : char;
   in     a1_2  : int;
   inout  a2    : int;
   in     p3o   : mach_port_t = MACH_MSG_TYPE_PORT_SEND;
   in     l3o   : lli_t;
   in     l3c   : lli_t;
   out    r     : char
)
```

In order to support inheritance in remote invocations, the *message base id* of the subsystem is given by the number of non-pure virtual methods defined in the direct and indirect base classes of the subsystem's corresponding class. Note that multiple inheritance is not supported.

The subsystem defines a MiG `routine` per each method in the original class. The routine has the same parameters as the method but object references are expanded into their "on-the-wire" representation and structures are recursively decomposed into their constituent elements. The method's return value is also added as a routine parameter.

Parameters passed by value are mapped onto MiG `in` parameters. If they are passed by pointer they are mapped onto `inout` parameters, to cover their worst case use. Note that call-by-value semantics are respected. However, call-by-reference is only respected with object references; with basic types, structures and arrays it can only be emulated by call-by-value-return.

Each routine's name is obtained by concatenating the class name, the method name and the method signature. This is needed to ensure the name resolution rules of C++.

The client proxy class generated in this example is as follows:

```
class C_aClass : public C_bClass {
public:
   virtual char aMethod(aStruct, int*, object*);
};

char C_aClass :: aMethod(aStruct a1, int* a2, object* a3)
{
   char r;
   LLI_t l3o, l3c;                    // object and class LLI
   mach_port_t p3o;                   // object port
   mach_msg_type_name_t t3;           // port type
   XRef(a3, &p3o, &l3o, &l3c, &t3);
   mach_port_t p;
   getPort(this, p);
   handleError(call_aClassaMethodF7aStructPiP6object(p, a1.aChar, a1.anInt, a2,
                                         p3o, t3, l3o, l3c, &r));
   return r;
}
```

The methods of the client proxy class decompose structures into their constituent types (and later compose them if the structures are passed by pointer or by reference). This step is important to guarantee the correct typing of the message which, according to the Mach philosophy, can be used to handle different data representations with a receiver-makes-it-right policy. This decomposition also eliminates dependencies on the compiler "alignment inside structures" rules. Finally, this analysis is important to detect embedded object references or pointers which must be handled separately.

The macro `getPort` retrieves the remote object's port name from the proxy field of the client proxy's RTH.

`handleError` is a macro which calls the method `handleChildDeath()` in the event of failure of the context where the remote object was mapped. This method is defined in the base class `C_object`. It tries to map the object locally or obtain a send right to the new object port. It can be redefined to provide a per-class or per-object child-death handler.

This example's server proxy class is as follows:

```
class S_aClass : public S_bClass {
public:
   virtual void dispatch(mach_msg_header_t*, mach_msg_header_t*);
};

void S_aClass :: dispatch(mach_msg_header_t* in, mach_msg_header_t* out)
{
   if (M_aClass_server(in, out))
      return;
   else S_bClass::dispatch(in, out);
}

extern "C" {
kern_return_t
do_aClassaMethodF7aStructPiP6object(mach_port_t p, char a1_1, int a1_2,
int* a2, mach_port_t p3o, LLI_t l3o, LLI_t l3c, char* r)
{
   struct aStruct a1;
   a1.aChar = a1_1;
   a1.anInt = a1_2;
   object* a3;
   IRef(l3o, p3o, l3c, a3);
   aClass* obj;
   getObj(obj, p);
   *r = obj->aMethod(a1, a2, a3);
   return KERN_SUCCESS;
}
}
```

This class exports only the method `dispatch`, but its code module also defines the functions called by the MiG demultiplexer function. These functions are responsible for composing the structures received; importing references; invoking the corresponding method on the right object; decomposing the structures passed by pointer or reference and exporting references.

The object on which to invoke the method is selected using the macro `getObj`. As we have noted, the address of the server proxy's RTH matches the port name. Therefore, the macro locates the server proxy object using the port name and retrieves the object's reference from the server proxy's RTH.

One advantage of interposing proxy classes in remote invocations is that the programmer can change the default implementation to meet specific application needs [12] (e.g.

introduce caching for efficiency, encapsulate shared memory access and manage replication). Furthermore, different client proxy classes can be used to access the same remote object.

The layering of the proxy classes on top of MiG stubs eases the interconnection of MIKE's applications with existing servers. The server's client stub can be encapsulated in a client proxy class and the server can be handled as an always remote MIKE object. For example, references to the server "object" can be freely exchanged in invocations. In spite of this advantage, MiG stubs are large. The use of a dynamic marshalling and unmarshalling mechanism similar to the one used in [6] would halve proxy class sizes.

3.3.3 Inheritance in remote invocation

Inheritance in remote invocation is supported by the addition of two shadow replicas of the original class hierarchy, one for the server proxy classes and another for the client proxy's. The class hierarchy for the example we have been presenting is shown in figure 4.

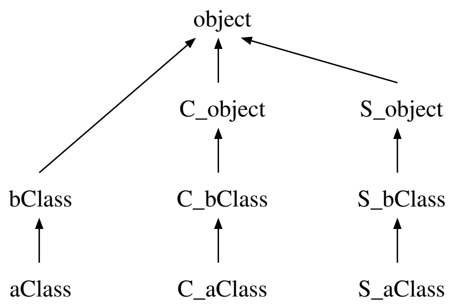

Figure 4: Class hierarchy which supports inheritance in remote invocation

The client proxy hierarchy ensures that they inherit the same interface as the original classes. The server proxy hierarchy guarantees that the base classes are all loaded when `dispatch` is invoked in a given class. As shown in this example, the method `dispatch` calls the MiG demultiplexer function. If the message identifier falls out of range, it calls the method `dispatch` of its base class.

This scheme only supports single inheritance and all methods must be virtual. Note that `C_aClass` can be used wherever `aClass` is expected without the need to define the common interface in a base class. This is possible because MIKE supports a late-binding mechanism similar to Smalltalk's, it would not work in plain C++.

3.3.4 The Dispatcher

Each context has an instance of class Dispatcher. Its function is to receive remote invocation requests and call the correct method on the correct object to service them. The Dispatcher manages a set of service threads that concurrently wait for remote invocation requests on a port set. All the object ports are inserted in this port set.

The number of service threads is adjusted dynamically to adapt to varying load. The `dispatcher` object has two counters – `totalThreads`, which counts the total number of service threads, and `freeThreads`, which counts the number of threads waiting for requests. These counters are updated concurrently by the threads under the protection of a lock.

Initially only one service thread is created. This thread executes the method `Dispatch`, a modified version of `mach_msg_server`. When a message is received `freeThreads` is decremented. If it becomes null and `totalThreads` is lower than a maximum value, a new thread is created to execute method `Dispatch` on the `dispatcher` object and `totalThreads` is incremented. Then the request is processed.

After processing the request `freeThreads` is incremented. If it reaches a given maximum and `totalThreads` is higher than a minimum value, the thread sends the reply and destroys itself. Otherwise it sends the reply and blocks waiting for a request using a combined send and receive operation for increased efficiency.

This simple scheme has the hysteresis needed to avoid oscillations and will avoid thread creation overhead except in abnormal load peaks.

The request dispatching is very simple. It is achieved with the following line of code:

```
((S_object *) Req->Head.msgh_local_port)->dispatch(&Req->Head, &Rep->Head);
```

Where `Req` is the buffer with the request message and `Rep` the buffer where the reply message will be placed. The dispatching takes advantage of C++'s polymorphism and of port name and server proxy reference matching. Since the port set only contains object ports it is guaranteed that the obtained reference always points to a valid object.

3.4 Garbage collection

MIKE uses a slightly modified version of IK's garbage collector [5]. Small objects are recycled using an algorithm similar to generation scavenging. Garbage collection of large objects and run time headers is based on an incremental mark-and-sweep algorithm. Only volatile objects are recycled.

The garbage collector roots are the register states and stacks of all the threads in the context, the references in the instance data of persistent objects with an LLI and a set of object references called `refSet`. Register states can easily be obtained using `thread_get_state`, but finding the used portion of each thread's stack can be more difficult. When a thread is preempted while executing emulation library code, the stack pointer value saved in the register state points to an emulation library stack. The real stack pointer value is saved at the bottom of this stack. This situation must be detected and handled correctly.

Each context has a collector thread. This thread executes the collection algorithm independently from other contexts. Synchronization between mutator and collector threads is based on a stop-the-world policy. The collector thread suspends all other context threads before executing the algorithm. Mutual exclusion is used to guarantee that threads are suspended in a safe state. The collector thread acquires the locks which protect sensitive data structures before suspending the other threads.

Distributed garbage collection combines Mach's built-in port reference counting mechanism with the local garbage collection. When a server proxy is associated with an object (in `XRef`) both references are inserted in the `refSet`. Then a no-more-senders notification for the object port is requested to the kernel. The context who imports the reference gets a send right to the object port and an associated client proxy object.

Eventually, the client proxy object stops being referenced and is recycled by the local collector. When this happens the send right is destroyed. If it is the last client proxy the context where the object is mapped will receive a no-more-senders notification. The notification handler compares the current receive right *make send count* with the one in the notification message [8]. This comparison is made under the protection of the object lock to ensure that no references are exported during the process. If the *make send counts* differ a new notification is requested. Otherwise, the server proxy and object references are removed from `refSet` and they become subject to the normal garbage collection process. The receive right is destroyed when the server proxy is garbage collected.

MIKE's distributed garbage collection, like any other simple distributed reference counting algorithm, does not collect distributed cycles.

3.5 Object persistency

From the programmer's point of view there is a one-level store, all objects are manipulated in the same way whether they are persistent or volatile, mapped or on disk. Persistency is a dynamic attribute.

Objects are created volatile. They are promoted to persistent when they become reachable from a reference on persistent store. An activity can explicitly save a reference to an object on persistent store using the name service to assign a name to the object.

A persistent object's image can be saved explicitly invoking the method `flush` on the object or implicitly when the context where it was mapped terminates. When the image of a persistent object is saved all the objects reachable from its references are also saved.

A persistent object's reference can be obtained given its name, using the name service, and then the object can be invoked and its references followed transparently. A persistent object is mapped in the first context that invokes it.

If a persistent object ceases to be referenced from persistent store it is turned volatile and can eventually be garbage collected.

Several system components cooperate to offer this view to the programmer. The name servers offer the persistent name service and the storage servers and the run time support library cooperate to save and retrieve persistent object's images.

3.5.1 Persistent object naming and clustering

MIKE's persistent references are called LLI. The LLI is a tuple with three components (<*SSid, BGN, GN*>). The *SSid* (Storage System identifier – 16 bits) identifies the storage system container where the object's persistent image is stored and the storage server responsible for the object. The *BGN* (Base Generation Number – 32 bits) and the *GN* (Generation Number – 16 bits) identify the object within a container.

Since there is a potentially very large number of persistent objects we do not assign an LLI to all of them. An LLI is assigned to an object (if it does not already have one) only in the following situations:

- The name service is used to assign a persistent name to the object.
- A reference to an object is exported to another context. If this object was volatile and later ceases to be distributed (which is detected through the distributed garbage collection) the assignment is invalidated.
- If, when saving persistent objects, it is found that an object is reachable from more than one object with an LLI.

In our platform persistent objects tend to be fine grained, around 48 bytes average size, so we use an object clustering technique to obtain larger grained entities called clusters. These entities are then manipulated by the storage system.

A cluster is a subgraph of the global persistent objects graph. In this subgraph there is only one object with an LLI – the head of the cluster. The head of the cluster is the only object directly referenced from the exterior of the cluster and so the LLI of the head identifies the cluster.

The tail of the cluster is composed of all the objects in the private subgraph of the head, that is, those only accessible from the exterior of the cluster through references in the head's instance data. The objects in the tail are stored in depth first order.

In the current implementation the clusters are rebuilt by the run time support library every time the head is flushed or the context where they were mapped terminates. During this process object references are translated into cluster offsets, if they point to an object in the same cluster, or into LLI, otherwise. Then the clusters are delivered to the storage system which saves them on disk.

When a method is invoked on the head of an unmapped cluster the run time library asks the storage service to map the cluster given the head's LLI. If the request is successful the cluster is mapped and references are converted from their disk representation to pointers in a lazy way – only when each object is invoked.

3.5.2 The storage system

Clusters are stored in a distributed persistent object space partitioned into a set of storage system containers (Unix directories). Each container is managed by a different storage server, which runs at the node where the file system resides. Each user has a container assigned, all persistent object clusters created by this user's applications are stored in that container. Each cluster is stored in a separate file.

The storage servers are multi-threaded. There is a set of service threads waiting for requests in a service port. The service port is registered in the netname server [2] as "SS*ssid*", where *ssid* is the container's storage system identifier.

Each storage server maintains an object location table. This hash table is used to obtain binding information for mapped persistent objects. Each entry in this table stores the object's LLI, a send right to its port, its client proxy class' LLI and a send right to the object port of the context where the object is mapped. The storage server uses Mach's port destroyed notifications to clear entries in this table when the contexts where the objects were mapped terminate.

A context interacts with the servers through a proxy layer, which encapsulates MiG stubs. This layer is part of the run time support library. When an operation is requested on an object, the proxy layer extracts the field *SSid* from the object's LLI and uses it to obtain the server's service port from the netname server. After acquiring the service port the operation is requested to the server using RPC. The service port is cached for efficiency.

When a context requests the first LLI, the proxy layer obtains a *BGN* value from its storage server. The context can use this value to create 2^{16} LLI without communicating

[2] In the non-NORMA kernels the netname service is part of the netmessage server. In NORMA the netname service is a modified version of *snames* which offers the old netname server interface.

with the server. The context port and the *BGN* are registered in the storage server's object location table. If the context exhausts the 2^{16} LLI the proxy layer requests a new *BGN*.

The proxy layer requests the appropriate server to map a cluster as a consequence of an object fault on the cluster's head. When a storage server receives a map request, it searches its object location table to find if the cluster was mapped. If the cluster was not mapped, the server reads the cluster from disk and passes it to the context in an IPC message. The server also registers the object's LLI and the context's port in the object location table. If the cluster was mapped, the storage server returns a send right to the object's port and the LLI of its client proxy class. If this information is not present in the object location table, it is obtained from the context where the object was mapped. If needed the context will allocate a server proxy and a port for the object.

When a cluster is flushed or unmapped it is sent to the appropriate storage server in an IPC message. The server writes the new cluster state on disk. If the cluster is being unmapped its entry is removed from the object location table. Concurrent accesses to an object's persistent image are serialized using mutual exclusion primitives to synchronize the storage server's service threads.

4 Evaluation

This section evaluates some key aspects of MIKE's performance and compares MIKE with the versions of IK which only use traditional UNIX abstractions.

The test environment was composed of one 33 MHz i386 Intel 303 and a 33 MHz i486 Topzen, connected by a 10 Mbit *Ethernet*, running MK77 [3] in the intra-node tests and NORMA 12 [4], in the inter-node tests. In the inter-node tests the client was in the i386 machine.

4.1 Object invocation performance

This section evaluates the performance of object invocation. The benchmark program is very simple – one object invokes an empty method on another object. The tables present the method signatures followed by the elapsed times. In the method signatures, `object*` is a reference to a MIKE object. The values were determined by executing the test in a tight loop 100,000 times and computing the average elapsed time of each pass through the loop.

Table 1 compares MIKE's intra-context object invocation cost with a virtual method invocation in C++.

	MIKE	**C++**
`void aClass::f(int *)`	6.8 μs	2.6 μs

Table 1: Intra-context invocation elapsed times.

[3] Compiled with STD+WS.

[4] Compiled with STD+WS+assert+lineno+NORMA+norma_ether.

C++ invocation is 62% faster for two reasons. Firstly, C++ uses dynamic binding, which is more efficient than the late binding used in MIKE. Secondly, C++'s binding mechanism is reentrant. MIKE uses a per-class method cache shared by all threads. Accesses to this cache are synchronized, adding approximately 2μs to the invocation cost. A per-thread global cache could be more efficient, avoiding the need to synchronize accesses.

Table 2 presents remote invocation costs. Mapped means that the referenced object or an associated client proxy is mapped locally. With unmapped persistent object's references an optimization is used (see section 3.2.4).

		Intra-node	**Inter-node**
`void aClass::f(int *)`		315 μs	3053 μs
`void aClass::f(object *)`	unmapped	469 μs	3344 μs
`void aClass::f(object *)`	mapped	653 μs	3560 μs

Table 2: Remote invocation elapsed times.

Passing an object reference is more expensive than passing a pointer to an integer due to the costs of exporting and importing the reference. It is around 28% cheaper to export an unmapped object's reference because sending a port is relatively expensive.

The results expose the problem of performance transparency. Invocation costs grow significantly with the "distance" between the invoker and the invoked object. Therefore, the programmer (or the platform) should locate closely related objects together to achieve good performance.

Table 3 compares the time elapsed in a simple RPC, based directly in MiG stubs, with the time spent in invoking a method with similar parameters on a remote object.

`void f(int *)`	270 μs
`void aClass::f(int *)`	315 μs

Table 3: Intra-node remote invocation versus simply calling MiG stubs.

MIKE's remote invocation mechanism adds three object invocations to the MiG RPC code path – the invocation of the client proxy, the invocation of method `dispatch` in the server proxy and the invocation of the object. This combined with the overhead of dynamically managing the number of dispatcher threads adds 17% to the cost of a simple intra-node MiG RPC. We believe this extra cost is well justified by the functionality provided.

4.2 Implementation on UNIX vs. implementation on Mach

This section highlights the major benefits of using Mach to support a distributed object-oriented programming platform. It compares MIKE with the versions of IK that only use traditional UNIX abstractions.

In IK threads are implemented by user level code without kernel support, which has the usual advantages and disadvantages [14]. Since IK controls the thread scheduling, preemption can only occur in well known points. This renders synchronization unnecessary in most accesses to shared data structures. In particular, there is no need to synchronize accesses to the method's cache, which makes IK's intra-context invocation faster than MIKE's.

One of the problems with IK's threads is the support for blocking system calls. IK redefines the most common blocking system calls to implement preemption. This mechanism is based on the `SIGIO` signal and the `select` system call and it introduces a significant performance overhead. It is responsible for 30% of the total cost of an inter-node remote invocation between a Sparc 10/20 and a Sparc 10/30, where the SunOS 4.1.3 LWP thread package is used [13]. Furthermore, when a thread blocks in a page fault requiring I/O, the entire context blocks.

On the other hand, MIKE uses Cthreads supported by multiple kernel threads. These threads can handle blocking operations more efficiently, including page faults. Mach threads support parallelism inside a context, while IK must use several contexts to explore the parallelism offered by a shared memory multiprocessor.

IK uses BSD sockets to perform remote invocations, this ensures portability and interoperability, but MIKE's remote invocation is faster than IK's. The main reasons for this are the more efficient IPC and threads implementation on Mach. In particular, MIKE's intra-node remote invocation is ten times faster than IK's SunOS version on top of a Sparc 10/30 [13] and the inter-node remote invocation is 30% faster than in IK (in the same test configuration as described in the previous paragraphs). On the other hand, inter-node remote invocation supported by the netmessage server is very slow.

The use of Mach ports as distributed object references played a fundamental role in the implementation of MIKE's distributed garbage collector and child death detection mechanisms. Another important advantage of using Mach ports is the per-object protection by capabilities.

In IK the name service and storage system code is linked with the run time support library. IK relies on a distributed file system to name persistent objects and access their images. MIKE takes advantage of the efficient Mach IPC and uses a multi-server architecture. This eliminates the dependency on a distributed file system to access persistent objects. Furthermore, the multi-server architecture will hopefully ease MIKE's port to other operating system "personalities", because its run time support uses almost only Mach abstractions and UNIX dependencies are isolated in the storage and name servers.

5 Related work

This section compares MIKE with other systems that exploit micro-kernel functionality with similar goals. Namely COOL [2], the multi-server [6] and Casper [15]. The first one exploits the Chorus [10] functionality and the last two exploit the facilities offered by Mach.

MIKE's distributed object sharing support is similar to the one in the multi-server. The two systems use a function shipping model implemented using Mach inter-process communication. Casper and COOL use data-shipping to share objects between contexts. Both systems use a page-based distributed shared memory similar to Li's [7], implemented with external pagers. COOL also supports an RPC based object sharing mechanism.

MIKE does not use a page-based distributed shared memory because of the well known false sharing problem. In fact, our average object size, 48 bytes, is much smaller than the unit of sharing, the 4 Kbytes page. We believe the unit of sharing must be the language level object or even fragments of it [4]. Casper and COOL use object clustering techniques to minimize the false sharing problem.

Another problem with Casper and COOL's distributed shared memory implementation is that they offer a sequentially consistent memory, using page invalidation to detect and classify memory accesses. This memory model is expensive and usually accesses must also be synchronized at an higher level.

Object naming in MIKE and the multi-server is similar. They both associate ports with distributed objects and use port capabilities as distributed object references. However, MIKE supports object persistency, which raises several problems (e.g. send rights must be complemented with persistent identifiers to reference potentially persistent objects).

Casper and COOL also support persistency, but they use virtual memory pointers as distributed and persistent object references. This approach eliminates the need for pointer swizzling and allows persistent objects to be loaded by the normal demand paging scheme. On the other hand, the systems must ensure that contexts use the same addresses for the same distributed objects and persistent object addresses must be allocated persistently. This can be expensive and in Casper it restricts persistent store size to the virtual address space size of the processors memory management unit (4 Gbyte), which is a severe restriction. With the advent of wide address space processors this restriction will become unimportant. COOL avoids this restriction by supporting a form of pointer swizzling.

Local garbage collection in MIKE is similar to the one in Casper, but the former recycles persistent objects. This task is simplified by the use of a single storage server. This centralized storage server prevents scalability. Like MIKE, the multi-server uses no-more-senders notifications to implement distributed garbage collection, but the multi-server also uses reference counting for local garbage collection.

The multi-server, COOL and MIKE use C++ to program applications. In Casper programmers use the Napier88 programming language. MIKE, COOL and Casper support distribution and persistency transparently, but in the multi-server the proxies must be hand coded and the programmer must specify the RPC messages using a subset of an IDL.

6 Conclusion

We described an object-oriented programming platform which supports transparent access to remote and persistent objects programmed in C++. This implementation runs on top of the Mach NORMA kernel and the BSD single server. It makes extensive use of Mach IPC and real multi-threading facilities. Since the run time support library is built using mostly the micro-kernel interface, the port to other operating system "personalities" should be easy.

The use of Mach IPC and threads was determinant to achieve good remote object invocation performance. Mach threads also allow MIKE to support parallelism inside a single address space.

The use of ports as distributed object references allows for an easy implementation of distributed garbage collection and child death detection mechanisms and provides per-object protection by capabilities. In spite of this advantages, the use of send rights as references to potentially persistent objects proved insufficient. To solve this problem and retain the advantages of using ports we complement send rights with a persistent global unique identifier.

The desire to support fine grained objects led us to the use of lazy evaluation techniques to minimize the overhead of associating a port per distributed object and using two "communication classes" per each original class. Ports are allocated in a lazy way and proxy classes are also loaded and linked lazyly.

The EC++ proxy layer on top of MiG makes remote invocation very flexible without degrading performance. Proxy classes are generated by a tool in a way transparent to the programmer. Nevertheless, the programmer can modify the generated code to meet specific application needs. This architecture also eases the interconnection of MIKE's applications with existing MiG servers.

We believe that our platform eases the development of distributed applications, but there is still a lot of work to do in hidding the lack of performance transparency and the partial failures from cooperating application components. Therefore, we are currently investigating data shipping based distributed object sharing techniques combined with group communication tools.

Acknowledgements

We would like to thank the members of our group at INESC, in alphabetical order, Adriano Couto, Paulo Ferreira, Paulo Guedes, Cristina Lopes, David Matos, João Pereira, José Pereira, Mario Reis, Manuel Sequeira, António Silva and André Zúquete for their work in the IK platform. Special thanks go to Paulo Guedes and Manuel Sequeira for their careful reading of this manuscript and for their many suggestions that led to its improvement.

Availability

MIKE and the distributed ray-tracer are available on request from the authors. E-mail can be sent to `ik-staff@sabrina.inesc.pt`.

References

[1] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A New Kernel Foundation for UNIX Development. In *Proceedings of Summer Usenix*, July 1986.

[2] P. Amaral, C. Jacquemot, P. Jensen, R. Lea, and A. Mirowksi. Transparent Object Migration in COOL-2. In *Proceedings of Workshop on Dynamic Object Placement and Load-Balancing in Parallel and Distributed Systems, ECOOP'92*, June 1992.

[3] Richard P. Draves. A Revised IPC Interface. In *Proceedings of the USENIX Mach Conference*, October 1990.

[4] Michael Feeley and Henry Levy. Distributed Shared Memory with Versioned Objects. In *Proceedings of OOPSLA'92*, pages 247–262, Vancouver, Canada, October 1992.

[5] Paulo Ferreira. Reclaiming Storage in an Object Oriented Platform Supporting Extended C++ and Objective-C Applications. In *Proceedings of the International Workshop on Object Orientation in Operating Systems - IEEE*, Palo-Alto, October 1991.

[6] Paulo Guedes and Daniel Julin. Writing a Client-Server Application in C++. In *Proceedings of the USENIX C++ conference*, August 1992.

[7] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 17(4):321–359, November 1989.

[8] Keith Loepere. *Mach 3 Server Writer's Guide.* Open Software Foundation, January 1992.

[9] José Alves Marques and Paulo Guedes. Extending the Operating System to Support an Object-Oriented Environment. In *Proceedings of OOPSLA '89*, pages 113–122, New Orleans, Louisiana, October 1989.

[10] Mark Rozier, Vadim Abrozimov, Francois Armand, Ivan Boule, Frédé Hermann, Michel Gien, Mark Guillemont, Claude Kaiser, Pierre Léonard, Sylvain Langlois, and Willi Neuhauser. Chorus Distributed Operating System. In *Computing Systems*, volume 1, pages 304–370, December 1988.

[11] Manuel Sequeira and José Alves Marques. Can C++ be Used for Programming Distributed and Persistent Objects? In *Proceedings of the International Workshop on Object Orientation in Operating Systems - IEEE*, Palo Alto, October 1991.

[12] Marc Shapiro. Structure and Encapsulation in Distributed Systems: The Proxy Principle. In *The 6th International Conference on Distributed Computer Systems*, pages 198–204, Cambridge, Mass., 1986. IEEE.

[13] Pedro Sousa, Manuel Sequeira, André Zúquete, Paulo Guedes, and José Alves Marques. The IK Distributed and Persistent Platform — Overview and Evaluation. Technical report, INESC, February 1993.

[14] Avadis Tevanian, Richard F. Rashid, David B. Golub, David L. Black, Eric Cooper, and Michael Young. Mach Threads and the Unix Kernel: The Battle for Control. Technical Report CMU-CS-87-149, Department of Computer Science, Carnegie Mellon University, August 1987.

[15] F. Vaughan, T. Basso, A. Dearle, C. Marlin, and C. Barter. Casper: a Cached Architecture Supporting Persistence. *Computing Systems*, 5(3):337–359, 1992.