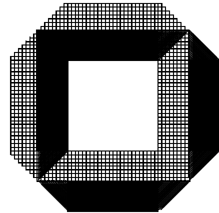
# Object Migration in Non-Monolithic Distributed Applications

O. Ciupke, D. Kottmann,

H.-D. Walter

Interner Bericht 43/95



Universität Karlsruhe

Fakultät für Informatik

(Submitted for publication)

## Abstract

Object migration is usually applied to optimize distributed monolithic systems. In this paper, the authors investigate whether object migration can also be utilized in cooperative systems which consist of autonomous components.

We show that object migration policies will not always optimize system performance. Rather they can reduce it drastically if different components apply these policies concurrently.

Conventional run-time support for linguistic primitives which are usually used to express migration policies is adapted to cooperative systems. We show that two novel approaches, place-policy and reduction of attachment-transitiveness, can counter the degradation caused by conflicting policies. In order to restrict attachment-transitiveness we introduce dynamic relationships called *alliances* between objects which explicitly define cooperation contexts.

The effects of these modifications are evaluated by simulation.

# 1 Introduction

In most modern application areas — take office or factory automation as examples — many independent components cooperate to fulfill a common task. They do so naturally in a distributed environment. But most often the components are not developed from scratch, but introduced in an evolutionary manner. Often components are applications themselves. It is an illusion to assume that there internal structure and external behavior is compliant because they are normally developed by independent teams or even purchased from off-the-shelf. However, they have to work together in one distributed system. In addition they must cooperate, which normally means that they share a subset of common data.

As the behavior of a distributed system is the net effect of the local behavior of independent components, the simple approach to map distribution entities statically to one node generally degrades quality-parameters like performance or fault-tolerance[1]. One depends on mechanisms like replication (cf. [Jal94]), fragmentation (cf. [MGL*94]) or object-migration (cf. [JLH*88]) to counter this degradation.

By their very nature, replication and fragmentation have always been discussed in the context of parallel accesses from different nodes. In case of object migration, parallel accesses are conventionally only treated for the case of immutable objects. Moving a static object simply creates a copy.

When independently developed components work together, we call the resulting system a *non-monolithic application*. Now the question arises whether is is possible to use distribution mechanisms in such applications to improve the quality of operation. The basic effect of letting independent components use mechanisms in one distributed system with shared objects basically means that there is unsynchronized concurrency. As long as the distribution mechanisms in such systems are not tailored according to this observation, they will useless or even worse detrimental. In this paper, it is shown that this is a particular danger if one employs object migration. Further, we will develop system-level mechanisms to counter this phenomenon.

The paper is organized as follows. In Section 2 we introduce the common notion

[1]The consequence is Leslie Lamport's definition of a distributed system: "'You know you have one when the crash of a computer you've never heard of stops you from getting any work done"'[Sch93]

of object migration, show what the term migration policies means, and separate the prerequisites that underly the conventional linguistic support for object systems that support migration. It will become clear that those approaches fail for non-monolithic applications. How conventional run-time support can be extended for these areas is the subject of Section 3. We identify two simple modifications: the place-policy and the reduction of attachment-transitiveness. For the latter we introduce dynamic relationships between objects called *alliances*. Alliances make cooperation contexts of objects explicit. The effects of these modifications are evaluated by simulation (Section 4). It is shown that the policies effectively counter the performance degradation imposed by conventional migration policies in non-monolithic applications. Finally, Section 5 concludes the paper and gives a brief outlook.

# 2 Object Migration

## 2.1 Distributed Object-Oriented Systems

Objects have a well-defined interface consisting of set of methods which can be invoked by clients. They hide the implementation of their methods, encapsulate their state and have no state in common with other objects. Consequently, objects interact solely via message exchanges.

Hence, interactions among objects map readily to communication in a distributed system. This makes objects an ideal model on which to build distributed applications. To distribute objects, one needs only a level of indirection to trap remote invocations and forward them to the location of the remote object. The technical principles of such systems are well understood [ChC91] and need no further discussion.

## 2.2 Controlling Migration

As objects encapsulate their internal state, they are not only the ideal entity for distribution, but are also movable. This observation has lead to a wealth of systems which support mobile objects. To get an expression of the numerous alternatives, the reader is referred to the comparative studies of Borghoff [Bor92] and Nuttall [Nut94]. Those

studies are also a good source for references for many other mechanisms that have been employed to solve the numerous technical problems that come with mobile objects.

Basically, object migration is nothing else than a dumb tool. Benefits could only be drawn from this tool if it is used in a way compliant to the very goal one intends to achieve. Hence, not the tool, but the policy with which the tool is controlled is the central issue.

To justify this observation, look at the applications for which mobile objects could be exploited in a distributed world[2]:

- *load-sharing* — by moving objects around the system, one can take advantage of lightly used computers;
- *communication performance* — objects that interact intensively can be moved to the same node to reduce the communication cost for the duration of their interaction; and
- *availability* — objects can be moved to different nodes to provide better failure coverage.

Although, the list is a very small selection of the potential benefits, one can see that the different goals are not compatible in general. Note, for example, that availability calls for distributing objects, while performance calls for collocating them. What goal is followed, is subject to the stated policy. Most often, communication performance is the target to be achieved. Hence, we will also take this as the goal that should be achieved through migrating objects.

To separate mechanism from policy, systems that support object migration normally only comprise a small set of primitives as building blocks for more complex mechanisms for specific applications. This basic linguistic support for mobile objects normally comprises of the following primitives.

- *Fixing objects* — Some objects should not be able to migrate at all. This could either be a permanent or a transient property. In the former case, the property is often expressed as a type attribute in order to force all of its instances to be sedentary.

[2] The list is a subset of the more complete discussion of [JLH*88]. We selected the points which are commonly regarded as being of general importance.

The latter is mostly the consequence of run-time decisions, e.g., to avoid thrashing. Linguistic primitives normally contain means to **fix()**, **unfix()** and **refix()** objects.

- *Moving objects* — The basic building block to demand collocations or dislocations is some variant of the **migrate(O, target)** primitive. The target either names a node or another object. In the latter case the objects are collocated. To decide whether it is sensible to migrate objects, their location could be interfered through primitives, like **location_of()** or **is_resident()**.

- *Keeping objects together* — Explicit migration only moves the specified objects without giving the system a clue about the reason for the move. For example, the system cannot infer whether collocation is permanent or temporary, i.e. whether it should latch the migrated objects to the target object specified in the **migrate()**-primitive or not. Instead, this must be expressed by the application based on predicted usage patterns. Consequently, the linguistic support for migration often contains some means to **attach()** objects and to **detach()** them. The system guarantees that attached objects are kept together until they are explicitly detached again. Attachment is transitive. A prominent example for the use of this technique is to simply keep an object together with all the objects it references through its attributes.

## 2.3 Migration Policies

The primitives presented above can be used as building blocks for arbitrary control policies. In addition many languages contain primitives that imply standard policies which are simple enough to be of general use. Two prominent examples are the **move()** and the **visit()** primitives. A move is a purposeful migration that is associated with some other primitive of the language. A visit is the combination of a move and a migrate back.

Normally those primitives could be used in operation declarations to force parameter objects to come to the callee (and to go back after the operation completed in the visit case). An example for this so called call-by-move or call-by-visit policy is given in Figure 1[3].

[3] The syntax is taken from GOM (Generic Object Model) [KeM94], the language used in our project.

```
type tool supertype ANY is
    body [ ...]
    operations
        declare assign: visit job, move schedule  → bool;
        ...
        implementation ...
end type tool;
```

Figure 1: Conventional Call-By-Move.

```
var l: [element]
...
begin
    visit(l,self); !! Move the list to the local node
    forall o in l do
    ...
    endfor !! Processing completed
end !! List migrates back
```

Figure 2: Conventional way to control migration.

An example of a more sophisticated use of the visit policy is given in Figure 2 which is an adaption of an example of [Ach93]. A list of objects (written as [obj]) is processed inside a loop. As there may be many accesses to an individual object in the list, the loop is enclosed in a block (begin/end) at whose beginning the list is forced to visit the processing node, and is kept there until the block has been completed.

Those primitive policies have one advantage compared to the basic **migrate()**-primitive: they carry semantics. A **migrate($O_x$,$N_1$)** only tells the system that the application wants to have object $O_x$ at node $N_1$ for some reason. Conversely, a **move()** always has a time during which is valid. In case of the call-by-move this is the time needed to process the remote call, in case of the block it it the time the system spends in processing the instructions inside this block. The semantics of the move primitive is related to this time span — the programmer tells the system that the cost to migrate the named object is less than the cost to use the object remotely during the validity of the

To take a look at the classic call-by-move syntax, the reader is once more referred to [JLH*88]

move primitive.

## 2.4 Underlying Premises

Although, the presented primitives are a powerful tool to express control policies their use relies on two implicit assumptions:

- *Objects know their future communication patterns* — If this assumption does not hold, there is no base for any migration decision. Hence, an object should at least know the set of its communication partners, if it wants to interfere the actual patterns. Better though, the object should know exactly how the cooperation with its partners will develop.

- *All objects are trusted* — Any object may call for arbitrary attachments or fixings. Hence, no object can exert control over what other objects it is attached to or whether it is fixed at the moment. In order to make sure that all policies expressed by individual objects sum up to a sensible overall behavior, all objects are assumed to behave in a reasonable fair way.

These assumptions are appropriate for monolithic distributed applications that are set up by a single programmer or a small, closely-knit design team. In a world of autonomously developed objects or subsystems that live together and share some common objects which are service providers, these assumptions generally do not hold. For example, an object does not know about all its (transitive) attached partners, as any object may invoke the **attach()**-primitive with arbitrary arguments. Consequently, it may continuously underestimate the effect of an issued **migrate()**-primitive by assuming that fewer objects are clustered together than actually result from following all transitive attachments. Additionally, some implementors may behave completely egoistic to tilt the system towards good behavior for their own application that only accounts for a small subset of the overall activity in the system.

# 3 Run-Time-Support for Migration in Non-Monolithic Applications

In this section we present remedies for the discussed shortcomings of conventional migration support in non-monolithic environments. First we present how system support for non-monolithic environments could be integrated in a distributed object oriented system. The cornerstone is the use of the conventional linguistic support for object migration, but to reinterpret the primitives in case of conflicts. We present two approaches. The first one, the substitution of migration by transient placement, is completely transparent for the programmer. The second one relies on means to define cooperative subpopulation of objects. This is treated as an add-on to existing object systems that can be included without changing the operations of objects.

## 3.1 System Model

In distributed object-oriented systems, calls to objects are trapped, linearized and forwarded to the current location of callee. There they are delinearized to enable a conventional invocation. One common mechanism for this is the use of proxy-objects that serve as placeholders for remote objects and perform the discussed linearization and forwarding. (cf. [Ach93, ScM93]).

This procedure is followed for conventional calls and for migration calls. The latter are not transformed in an invocation, but interpreted by the run-time system at the node of the callee. This is the place that has to be modified to include support for non-monolithic environments. This is done instead of simply executing the request and transforming the object to its new place. The procedure for proxies is depicted in Figure 3. Note that this basic model normally does not introduce additional remote operations, as everything is performed locally at the callee.

## 3.2 Substituting Migration by Transient Placement

Placement is a simple policy to cope with concurrent **move()**-requests from different nodes. A **move()** request is as usual forwarded to current location of the object. When

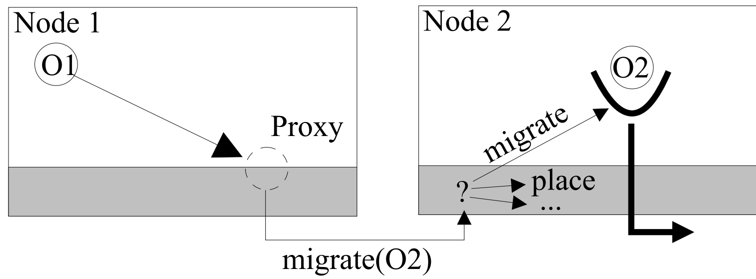



Figure 3: Including run-time support for non-monolithic environments.

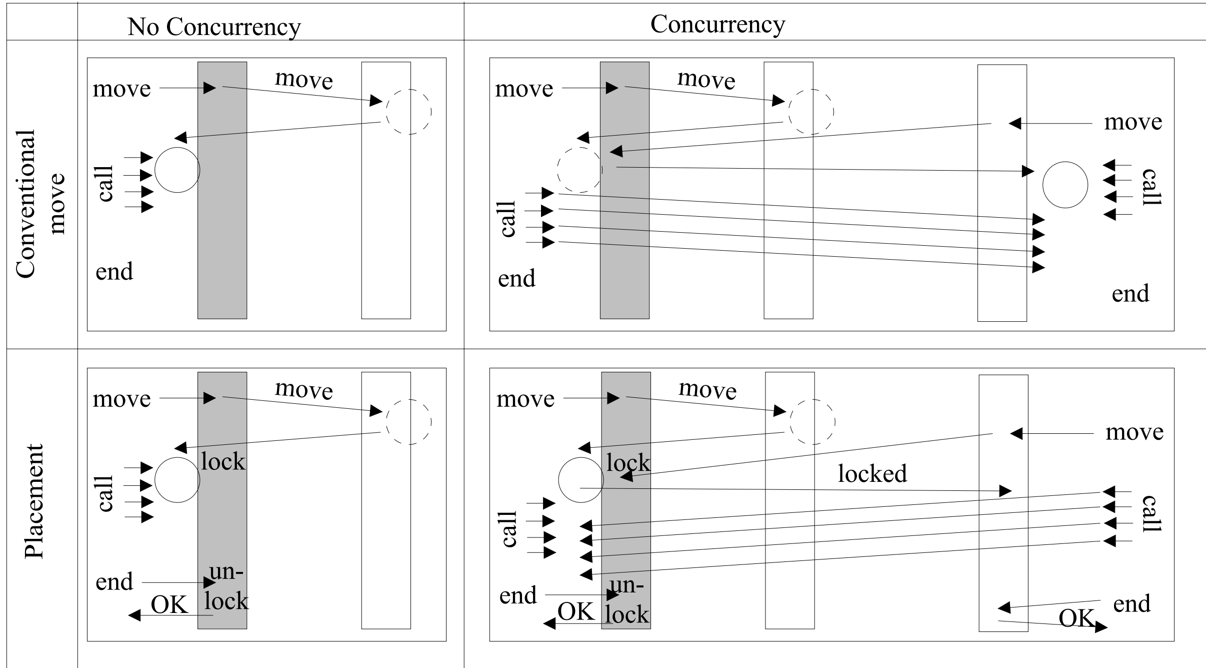



Figure 4: Successful and Unsuccessful placement.

no concurrent move is operating on the object, the primitive is executed as in conventional systems and the object is transferred to the caller. As soon as it arrives, the object is *locked*. A locked object is sedentary as long as the block or operation completes to which the **move()**-primitive is tied to. This is indicated to the run-time system with an **end**-request.

The resulting behavior is contrasted to conventional moves in Figure 4. Instead of transferring the object twice it is only moved once. Note that no additional remote operations are required to realize the new policy. The **end**-request is always a local

operation. In case the move was successful, the lock is removed. When the object is locked, the conflicting move-request returns an indication, the further calls at this node are forwarded to the object and the **end**-request is simply ignored, as nothing has to be done.

To see why this is approach is superior to the conventional one, let `C` denote the cost of a remote invocation message, `N` the number of calls to the object inside the move-block, and `M` the cost of a migration. The latter depends on the size of the object. But naturally `M` $>$ `C`. We assume that the programmer used the **move()**-primitive in a sensible way, which means that `N*C` $>$ `M`. In the depicted example for concurrency, the overall cost for the place-policy is `M + (2*N + 1)*C`, as a invocations comprise of a call and a result message. The worst case for the conventional policy occurs, when the second **move()**-request arrives before the invoker of the first request has performed any calls. The overall cost `2*M + (2*N + 2)` is worse than the cost of the place-policy.

We compare the two approaches in detail in Section 4.2, using a simulation model.

## 3.3 Exploiting Dynamic Information

The conventional move-policy is an aggressive migration-policy. On the other hand, placement can be seen as an conservative policy. Both operations do not exploit information about the set of users of an object, besides the information that an object is still in use by a move-block.

Those two policies are extremes of a continuum. Between them are policies that record information about the set of the current users of an object. To manage this, two additional requirements arise.

1. The run-time system has to collect additional data about the users. As this information has to be collocated with the object to enable its evaluation, the size of data that has to be transferred when migrating an object increases. Hence, such policies are clearly unpromising for small objects, because the additional information might grow to the multiple size of the object itself.

2. The information has to be kept up to date. Hence, all **move**- and **end**-primitives have to be forwarded to the location of the object. Thus, the advantage that the

place-policy of Section 3.2 does not increase the number of remote operations does not hold any longer.

Both requirements impose additional run-time overhead compared to the two simple policies. On the other hand, more intelligence can be coded into those policies. Take as an example the policy to locate an object at the node that issued the most concurrent **move()**-primitives. Whether the benefit of such policies outweighs their overhead is discussed in Section 4.3. There only the benefits are measured to keep the results clearly comparable to the simple policies. As we will see in this section, even in the absence of their overhead, the intelligent policies only provide a marginal improvement compared to the place-policy. Hence, situations in which they can be used are rare.

## 3.4 Keeping Objects Together

In the following, we use the **attach()**-primitive of Section 2.2 as the linguistic support to keep objects together. Attachments issued from different applications in a non-monolithic environment are harmless, as long as the same set of objects are kept together. This is certainly true when the applications have similar usage patterns. As discussed in Section 2.4 different usage patterns may impose considerable overhead, as the applications base their decision to issue an **attach()**-primitive on incomplete information and thus underestimate the cost of migrations.

The effect of different usage-patterns is that each application attaches together a set of objects with which it will work. This set is called its *working set*. Migration decisions are based on the knowledge of the own working set. But in case of diverging usage patterns, the working sets of different applications overlap and are grouped together by the **attach()**-primitives. Actually those clustered working sets are moved when one of the applications demands a migration.

To keep the migration decisions of applications sensible, the working set that is actually moved has to be the same as the one, the migration decision is based on. To achieve this, we introduced in [LW94, KLM95] the notion of an *alliance*. In a nutshell an alliance defines a dynamic relationship between a set of cooperative objects. It restricts the interaction between the objects to those that contribute to the target of the cooperation defined

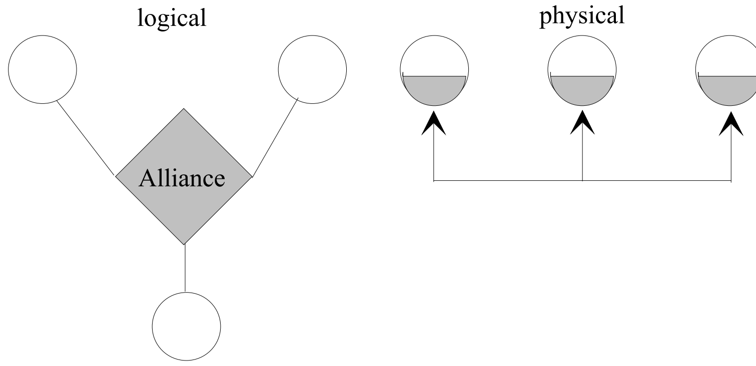



Figure 5: The Alliance Abstraction.

in the alliance. Thus, an alliance defines a cooperation-policy between a set of objects. Additionally, an alliance can define a distribution policy. We implemented a prototype to test the cooperation policies on top of Objectstore [LLO*] and a second one for distribution policies on top of DC++ [ScM93]. The resulting communication abstraction is depicted in Figure 5.

Some other authors recently proposed, too, to separate aspects of object cooperation either into a separate construct as, e.g., [YS94, HHG90, AFK+93], or by introducing special adapter- and mediator objects as, e.g., proposed in [GHJV94] which can be compared with alliances. But these approaches are mainly intended to support integration of objects with incompatible interfaces or to restrict method invocations in order to synchronize concurrent clients of a server object. None of them investigates the consequences of their approaches for distributed systems and only [AFK+93] treats concurrency.

The only detail of importance for the following discussion is that objects can be members of different alliances and that a primitive that controls migration can unanimously be related to one alliance. This can be exploited to let the actual working set that is migrated be the same as the one, migration decisions are based on. This is achieved in restricting the transitiveness of attachments to the ones defined by one alliance. Thus, attachments are *A-transitive*. Implicit migrations that result from attachments are restricted to the attachments of the alliance, the migration primitive was invoked in.

Another approach to restrict the moved working set to sensible dimensions, that does not rely on an additional construct like alliances, can be formed along the basic ideas of

the place-policy for movements of Section 3.2: *exclusive attachments*. An object is only allowed to be attached to one other object. All additional attachments for this object are ignored. When different applications have compliant usage patterns they will issue conforming attachments. Hence, the effect is the same as for conventional attachment. In case of diverging usage patterns, a fist-comes-first-served policy results that can be enhanced through exploiting dynamic information, similar to the extensions to the place-policy discussed in Section 3.3

No matter what policy is taken, the effect are disjunctive working-sets for applications. These effects are analyzed in Section 4.4. All in all, reducing attachments into cooperative subparts counters the negative effects of conflicting migration control in non-monolithic environments. The best performance is achieved when one combines the place-policy with attachment-reduction. Once again dynamic extensions to the place-policy have only negligible effects that are not worth their price.

# 4 Evaluation

The above discussed policies to control migration have been analyzed inside a simulation model. In the following the model is discussed and the key results for all policies are presented.

## 4.1 The Simulation Model

The communication structure among distributed objects is in general an arbitrary graph. The same is true for the network topology that interconnects the different nodes physically. Although it is merely a matter of complexity to model such complex structures into a simulation scenario, the obtained results would be hard to interpret. So we decided to use a simplified model to get clean results for the effects of our policies.

All the results that are presented in the next sections assume a fully connected network. We also performed simulations for other structures. But this had no effects on the results. In addition, the object system is assumed to run concurrently with other applications. Hence the network load imposed by the communication of objects does only contribute a small part to the overall load. Hence, saturation effects of networks like Ethernet

could be neglected. All in all, these assumptions result in a network model in which communication between objects is distributed around the same mean value for all nodes. In addition, we neglected the effects of different policies for object location, like name-server lookup [ChC91], forward addressing [JLH*88], broadcast [DLA*91] or immediate update [Dec86]. Hence, the time could be normalized so that a remote object invocation has an exponentially distributed duration of 1. This duration is valid for systems that do not support migration, because an object that is linearized and transferred over the net can not perform any operation until it is reinstalled at the target node. Hence migrations increase the mean duration, but in a way that cannot be fixed outside the simulation. Hence, a remote invocation in the simulation has a mean duration of 1 only when the callee currently resides on some node. When the object migrates at the moment of the invocation, the call is blocked until the object is operational once again. This increases the mean time of the communication. Local actions have been neglected, as they are normally about 4 orders of magnitude below the duration of a remote action [Ach93].

Because we are only interested into the effects our policy has for applications that use migrations, we only modeled the inter-object communication that occurs inside a move-block. The rest of the inter-object communication is part of the background load just as the other application in the net. Note that this decision strengthens our decision to neglect saturation effects of the network, as the relevant inter-object communication is only a subset of the overall inter-object communication. Further, we assumed that move-blocks are set up by the programmer in a sensible way. This means that a move block begins with a migration request and the net duration of all object invocations inside the block is bigger than the migration duration. Let `N` denote the number of invocations inside a move block and `M` the duration to migrate an object. As we normalized the mean duration for an object invocation to 1, a migration block is set up sensibly, when $\mathtt{N} > \mathtt{M}$. When a move block only uses one object, a move block is defined by the following parameters:

- the duration `M` to migrate the object,
- the number `N` of invocations to that object,
- the time $\mathtt{t}_i$ between two invocations of the object in the block, and

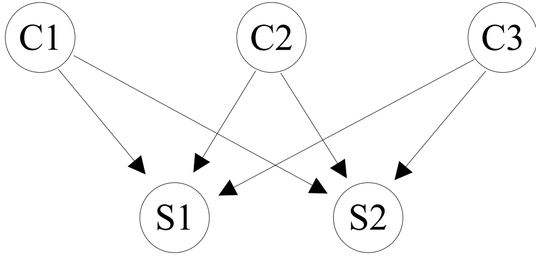



Figure 6: Basic Inter-Object Communication.

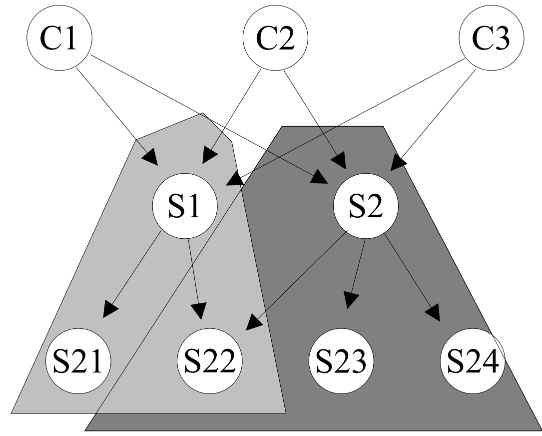



Figure 7: Inter-Object Communication for Attachments.

- the time $\mathtt{t}_m$ between the end of one move block and the start of the next one inside the same application.

All four parameters are assumed to be exponentially distributed. To keep the move block sensible, the mean value for the distribution of N has to be bigger than M. All times are given in multiples of the duration of an invocation. As a move block that uses multiple objects can be decomposed into multiple collocated move blocks that use only one object each, a move block in our model only uses a single object.

To keep the results interpretable, we assume a very simple inter-object communication structure: a client-server approach. Because each synchronous object invocation dynamically crates a client-server relationship between the two involved objects we simulated the basic building block of distributed applications. The resulting inter-object communication structure is depicted in Figure 6. Because clients are not invoked from other objects, there is no point in migrating them. Hence, they are sedentary. Only servers move during the simulation. Each client can communicate with each server. The move blocks operate inside the clients. Hence, concurrency and the rate of conflicting move-policies between different clients is incremented through two parameters: in incrementing the number of clients `N` or in decrementing the time between the move-blocks inside each client $\mathtt{t}_m$.

There is no point in introducing attachments to the basic inter-object communication structures, as the clients directly use each server. Attachments lead to implicit migrations of a working set. To model this, we extended the basic inter-object communication

| Parameter | Description | Distribution |
|---|---|---|
| D | Number of Nodes | fixed |
| C | Number of clients | fixed |
| $S_1$ | Number of 1st layer servers | fixed |
| $S_2$ | Number of 2nd layer servers | fixed |
| M | Migration duration for servers | fixed |
| N | Number of calls in a move-block | exp. |
| $t_i$ | Time between two calls in a block | exp. |
| $t_m$ | Time between two move blocks | exp. |
| — | Duration of a remote call | exp. (1) |

Table 1: Relevant Simulation Parameters

topology to the one depicted in Figure 7 for our simulative evaluation of the attachment-policies. The model comprises of two layers of server objects. The first one is directly used by the clients. Those servers use exactly the servers of the second layer belonging to the working set of this server. All server objects in one working set are attached together. We assume that these attachments are sensible. The effects discussed in Section 2.4 and 3.2 occur, when the working sets of servers partially overlap. The most important of these effects is that applications underestimate the cost of their migration decision, as their attachments are complemented by the ones of other applications.

To sum up, all relevant parameters for the simulation are given in table 1. All simulations have been run as long, as a confidence interval of 1% was reached with probability p=0.99.

## 4.2 Substituting Migration by Transient Placement

The place-policy was compared to the conventional migrate-policy and to a system that only consists of sedentary objects. Our claim is that we counter the performance degradation imposed by non-synchronized concurrent migration requests from different (parts of an) applications. We measured the performance variation that results from increasing

concurrency. Along the lines of our simulation model, two different sets of simulation have been performed.

1. A set with a fixed number of client and server objects where concurrency is increased by increasing the usage-frequency of objects inside the client.

2. A set with a fixed number of server objects, a fixed usage-frequency of each client, and an increasing number of clients.

One representative result from each set is presented in the next two subsections.

#### 4.2.1 Increasing the Usage-Frequency

The effect of increasing the usage frequency while using a fixed set of clients is depicted in Figure 8 that shows duration of invocations relative to the current usage-frequency. The duration is computed as the mean duration of an invocation plus the migration cost evenly distributed to the invocations belonging to that migration. Those numbers are separately displayed in Figure 10 and 11. The parameters of Figure 9 have been used. Note that only three clients have been active concurrently. Thus the mean duration of a call for sedentary nodes is 4/3, because it consists of a call and a result message and the change that the callee is local with respect to the caller is $1/C = 1/3$. The results show clearly that the migration can improve the performance in distributed systems. Furthermore, it can be seen that the place-policy performs better than the simple migration-policy in systems with unsynchronized access to a common set of objects from different clients.

The duration of invocations generally increase with concurrency. But Figure 8 shows that the duration decreases again, when concurrency approaches its maximum value. The answer for this behavior can be derived from the two detailed figures. Figure 10 shows that the duration of calls increases with concurrency, since the changes to migrate an object to the place of the caller and to perform all invocations locally decreases. On the other hand, the migration duration per invocation decreases at high concurrency levels, as shown in Figure 11. The reason is that the chance of finding that the callee is already collocated with the caller increase with concurrency.

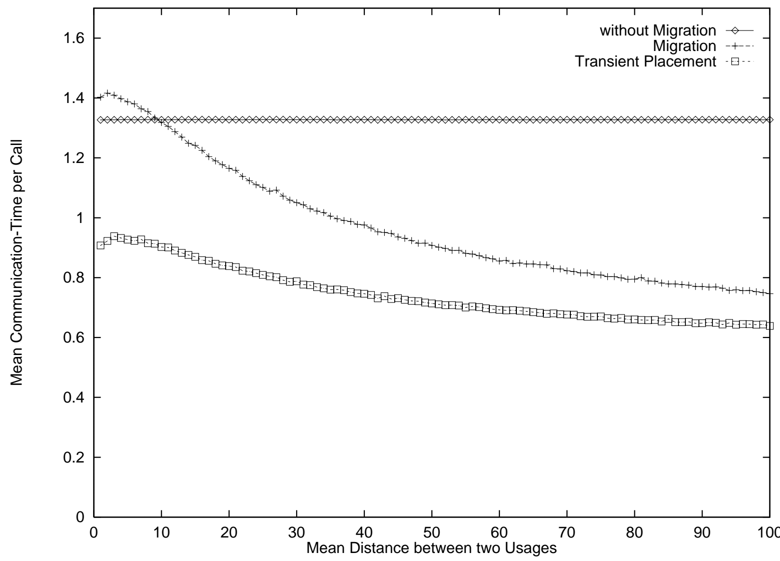



Figure 8: Increasing the Usage frequency.

| Para. | Distribution |
|---|---|
| D | 3 |
| C | 3 |
| $S_1$ | 3 |
| $S_2$ | 0 |
| M | 6 |
| N | mean(8) |
| $t_i$ | mean(1) |
| $t_m$ | variable |

Figure 9: Parameters.

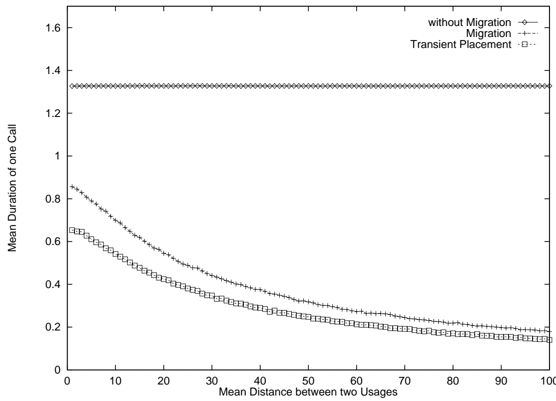



Figure 10: Duration of Invocations.

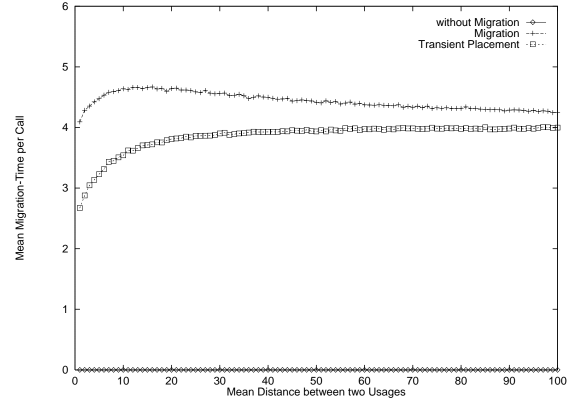



Figure 11: Migration-Load.

#### 4.2.2 Increasing the Number of Callers

The results discussed above have been obtained with a small fixed number of clients. Concurrency can also be increased in increasing the number of clients. This models the situation of hot-spot objects that are used by many clients. The common knowledge that it is better not to migrate such objects can clearly been interfered from Figure 12 (which was obtained with the parameters of Figure 13). The duration of calls linearly increases in the number of concurrent callers. The break-even point where migration gets worse than using fixed objects are 6 clients. Once again the place-policy is able to cope with the increased concurrency. The break even rises to 20 concurrent clients. It will be even bigger when the relation of object invocations inside a move-block to the migration duration (i.e.

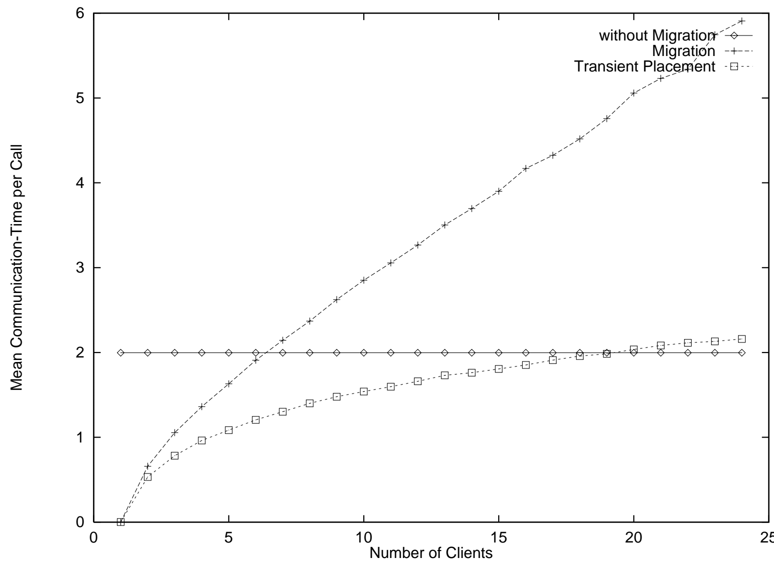



Figure 12: Increasing the Number of Clients.

| Para. | Distribution |
|---|---|
| D | 27 |
| C | variable |
| $S_1$ | 3 |
| $S_2$ | 0 |
| M | 6 |
| N | mean(8) |
| $t_i$ | mean(1) |
| $t_m$ | mean(30) |

Figure 13: Parameters.

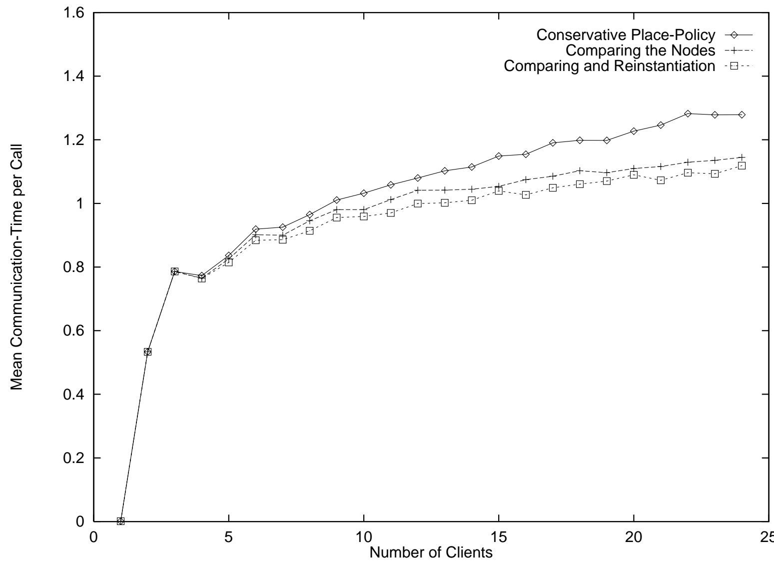



Figure 14: Increasing the Number of Clients.

| Para. | Distribution |
|---|---|
| D | 3 |
| C | variable |
| $S_1$ | 3 |
| $S_2$ | 0 |
| M | 6 |
| N | mean(8) |
| $t_i$ | mean(1) |
| $t_m$ | mean(30) |

Figure 15: Parameters.

`N/M`) increases. As the plot for the place-policy grows sublinearly in the number of clients and the growing rate decreases, an increase in `N/M` will have an over-proportional effect on the break even-point, contrasting to the basic migration policy.

## 4.3 Exploiting Dynamic Information

Figure 14 shows the mean communication-time per call using the conservative placement policy without any additional run-time information and the effects of two intelligent place-

ment strategies in relation to the number of concurrent clients. The used parameters are given in Figure 15.

The first strategy — comparing the nodes — bases its decision whether to migrate an object or not on the number of **move**-requests at one node. It tries to keep objects always at those nodes from where the most **move**-requests have been issued. For this it records **move**- and **end**-requests and the nodes where they have occurred. Now the situation may arise that a conflicting **move**-request has initially no effect on the location of the requested object but may lead to a migration at some point later if further **move**-requests are issued at the same node.

The second strategy — comparing and reinstantiation — treats **move**-requests in the same way than comparing the nodes. In addition objects may not only be migrated on **move**-requests but also on **end**-requests if an **end**-request leads to a situation that some other node holds a clear majority on open **move**-requests.

Figure 14 confirms our expectations from Section 3.3. Both strategies lead only to minor performance gains. Note that the necessary overhead to collect the dynamic information has been completely neglected in our simulation model. Hence, the improvement would be even smaller in real applications.

## 4.4 Keeping Objects Together

Figure 16 shows the effects of A-transitive attachments in combination with migration and transient placement. The underlying premise was that each alliance uses its associated objects in a distinct way. Thus, we considered only the worst-case. The parameters of this experiment are given in Figure 17.

We see that applying conventional migration together with unrestricted attachments has a devastating effect in non-monolithic environments. The more concurrent clients there are the more often they steal common servers from each other. This does not only leads to the migration of single objects but also to the migration of the transitive closure of all attached objects (servers of the second layer).

Transient placement combined with unrestricted attachment is a first improvement. The result is comparable to that shown in Figure 13. It can be explained by the conservative character of the policy: conflicting **move**-requests will not lead to the migration

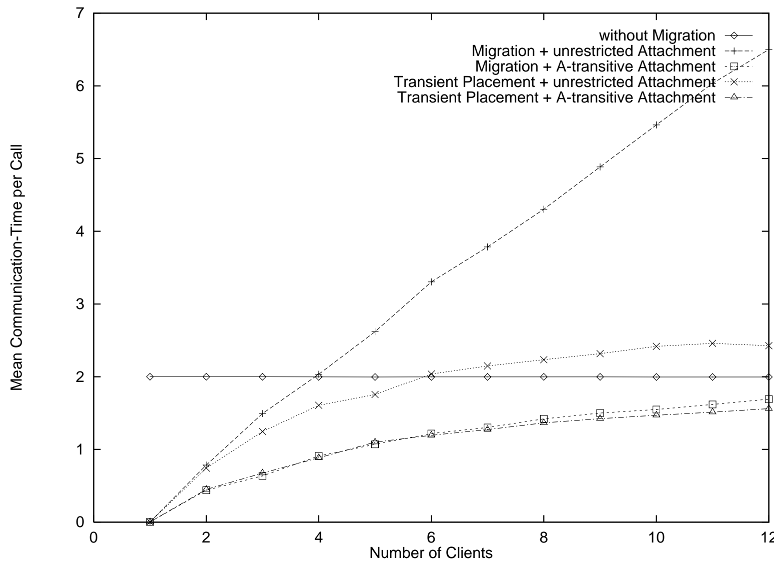



Figure 16: Increasing the Number of Clients.

| Para. | Distribution |
|---|---|
| D | 24 |
| C | variable |
| $S_1$ | 6 |
| $S_2$ | 6 |
| M | 6 |
| N | mean(6) |
| $t_i$ | mean(1) |
| $t_m$ | mean(30) |

Figure 17: Parameters.

of the requested object and, consequently, also not to the migration of objects attached to it.

The results of applying A-transitive attachment in combination with conventional migration and transient placement show that it can be worthwhile to attach objects which are in some way related by their predicted usage in non-monolithic systems, too. But this mechanism should only be applied if attachment is restricted to the context of an alliance if the usage-patterns for objects vary in different alliances. Remember hat the positive effect depends on the sensible usage of placement and attachment inside alliances.

# 5 Conclusion and Outlook

In this paper it has been shown that conventional object migration which has been proven a useful mechanism for monolithic systems can have negative effects in non-monolithic systems where autonomous components concurrently and cooperatively perform their tasks. The degradation mainly arises from conflicting policies which are applied by different objects which do not know what their colleagues in the system are doing.

Modifications of the semantics of common traditional linguistic support for object migration to adapt them to non-monolithic systems were proposed. We replaced the **move**-statement which normally leads to immediate migration of the specified object by

so-called transient placement which will only lead to migration if no other object has requested a **move** on this object before. In order to tell the system when an object is not needed any longer at a certain location an object additionally can issue an **end**-request. A simulation model confirmed the usefulness of transient placement. The simulation further indicated that even more intelligent policies to deal with conflicting **move**-requests which exploit run-time information will not be worth the overhead they cause and the additional implementation effort they require.

As a second modification to conventional mechanisms we proposed to restrict attachment to well-defined cooperation contexts. As an abstraction for these contexts we used alliances and restricted transitiveness of attachment to the set of their members. Restriction of alliance transitiveness leads to disjunctive working sets of applications. This prevents the costs for migration of being underestimated, because some other autonomous and concurrently active components have added elements to the transitive closure of a certain attachment. Again simulation confirmed our thesis.

In this paper we were concerned with object migration. But object migration is only one mechanism which can be used to enhance certain quality parameters of distributed systems. It seems worthwhile to investigate whether similar negative effects as we have shown for object migration arise for other mechanisms like replication and fragmentation if they are applied in non-monolithic systems, and if this is the case, how these mechanisms can be adapted to meet the special requirements of non-monolithic systems.

# References


[AFK+93] Agha G., Frølund S., Kim W., Panwar R., Patterson A., Sturman D.: *Abstraction and modularity mechanisms for concurrent computing*, In Agha G., Wegner P., and Yonezawa A., editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 3-21. MIT Press, 1993.

[Ach93] Achauer B.: *The DOWL Distributed Object-Oriented Language*, Communications of the ACM, 36(9), Sept. 1993, pp. 48-55

[Bor92] Borghoff, U. M.: *Catalogue of Distributed File/Operating Systems*, Springer-Verlag, 1992

[ChC91] Chin R. S., Chanson S. T.: *Distributed Object-Based Programming Systems*, ACM Computing Surveys, 23(1), March 1991, pp. 91-124

[Dec86] Decouchant D.: *Design of a Distributed Object Manager for the Smalltalk-80 System*, OOPSLA 86, ACM

[DLA*91] Dasgupta P., Le Blanc R. J. Jr., Appelbe W. F., Ramachandran U.: *the Clouds Distributed Operating System*, IEEE Computer, November 1991

[GHJV94] Gamma E., Helm R., Johnson R., Vlissides J.: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

[HHG90] Helm R., Holland I. M., Gangopadhyay D.: *Contracts: Specifying behavioral compositions in object-oriented systems*, IN: *Proc. of ECOOP/OOPSLA*, pages 169-180, 1990.

[Jal94] Jalote P.: *Fault Tolerance in Distributed Systems*, Prentice Hall, Englewood Cliffs, NJ, 1994

[JLH*88] Jul E., Levy H., Hutchinson N. Black A.: *Fine-Grained Mobility in the Emerald System*, ACM Trans. on Computer Systems, 6(1), Feb. 1988, pp. 109-133

[KeM94] Kemper A., Moerkotte G.: *Object-Oriented Database Management: Applications in Engineering and Computer Science*, Prentice Hall Inc., Englewood Cliffs, NJ., 1994

[KLM95] Kottmann D. A., Lockemann P. C., Walter H.-D.:*Multi-Object Cooperation in Distributed Object Bases*, Technical Report 95-16, Faculty of Computer Science, University of Karlsruhe, April 1995

[LLO*] Lamb C., Landis G., Orenstein J., Weinreb D.:*The Objectstore Database System*, Communications of the ACM, 34(10), October 1991, pp. 133-150

[LW94] Lockemann P. C, Walter H.-D.: *Activities in Object Bases*, IN: Paton N. W., Williams M. H.: *Rules in Database Systems*, pp. 3-22, Springer, 1994.

[MGL*94] Makpangou M., Gourhant Y., Le Narzul J.-P., Shapiro M.: *Fragmented Objects for Distributed Applications*, IN: Casavant T. L., Singhal M.: *Readings in Distributed Computing Systems*, IEEE Computer Society Press, Los Alamitos, 1994.

[Nut94] Nuttall M.: *Survey of Systems providing process or object migration*, Technical Report, Imperial College Research Report DoC94/10, Imperial College, London, UK, 1994

[Sch93] Schroeder M. D.: *A State-of-the-Art Distributed System: Computing with BOB*, IN: Mullender S. (Eds.): *Distributed Systems*, 2nd Edition, ACM Press Frontier Series, Addison-Wesley, 1993, pp. 1-16

[ScM93] Schill A. B., Mock M.U.: *DC++: Distributed Object-Oriented System Support on Top of OSF DCE*, Distributed Systems Engineering Journal, 1(2), 1993

[YS94] Yellin D. M., Strom R. E.: *Interfaces, protocols, and the semi-automatic construction of software adaptors*, IN: *Proc. of the ACM Conf. on Object-Oriented Programming Systems and Languages (OOPSLA)*, pages 176-190, Portland, Oregon, USA, Oct. 1994.