Migration in CORBA Component Model

Jacek Cała

AGH — University of Science and Technology
Department of Computer Science
jcala@agh.edu.pl

Abstract. Migration of running application code is considered a very attractive and desired mechanism to improve application performance, resource sharing, self-adaptability, etc. This mechanism seems to be even more important nowadays, considering the growing interest in the area of mobile computing and mobile networks.

This paper briefly presents a migration mechanism for a CORBA Component Model platform. We discuss general issues related to migration of running code, further elaborated in the context of CCM. We also propose an extension to the original CCM model which provides interfaces to implement migration.

The paper presents the most important problems which appeared during implementation of a prototype facility and it discusses possible solutions. One of the most fundamental issues related to mobility of running code is the *residual dependency problem*. The intention of the work is not to provide a solution to this (possibly unsolvable) problem, but to propose an approach which would make programmers aware of its existence. Thus, the paper allows readers to make more conscious decisions when designing their components. The paper ends with an evaluation of the prototype implementation on top of OpenCCM, an open source Java implementation of the CORBA Component Model.

1 Introduction

Migration of processes, tasks, objects, components or even whole operating systems during runtime is considered a very attractive and desired mechanism. Since the 1980s, there has been substantial interest in migration but, unfortunately, with very little use in real-world applications [1,2]. Today, however, as systems become more and more distributed in nature and with increasing interest in Component and Service Oriented Architectures (COA and SOA), migration mechanisms are more attractive than ever, since they enable better processing power exploitation, resource sharing, fault avoidance, mobile computing and self-adaptability.

Migration as a mechanism to facilitate dynamic load distribution may increase exploitation of available processing power by shifting a task from an overloaded node to another node, with sufficient CPU resources. It may also substantially reduce costs associated with frequent remote communication, improving effectiveness of a distributed system. Instead of calling remote objects, it is often more efficient to move one of the communicating sides directly to the other. The same strategy may be used to facilitate better resource sharing. If there is a node with a large amount of memory or specialized hardware devices, it might be useful to move a software component to this node in order to fully leverage its resources.

Moving instances of running code may also positively influence fault tolerance as well as system maintenance aspects. Given a migration facility, system administrators can move a running application to another node to perform maintenance tasks on the original host machine. Moreover, in more autonomous systems one can imagine that migration would be triggered by a fault detection mechanism whenever there is a suspicion of hardware failure.

Migration may also greatly support system self-adaptability, enabling reaction to changes in the environment e.g. appearance of a new mobile node. In addition, it may support deployment of an application according to changes in the environment. This aspect — support for dynamic and adaptive deployment of component-based applications in heterogeneous hardware and software domains — was the primary motivation behind the provision of a migration mechanism for a platform implementing the CORBA Component Model. The presented paper, however, does not focus on the deployment process itself but rather on issues directly related to the design and implementation of a CCM movement facility.

The paper is organized as follows. The following section covers work related to migration mechanisms, not limited to component architectures but more general in scope. Section 3 briefly introduces the fundamentals of migration mechanisms with relation to component environments. The next section presents the proposed extensions to the CORBA Component Model in order to facilitate migration. Section 5 depicts the internals of the adopted approach and presents solutions to the most important issues. In Section 6 we present evaluation tests of the prototype implementation. The work ends with conclusions and future development directions.

2 Related Work

The problem of migration of application code has already been addressed by many previous research projects and works such as [1], which gives a comprehensive report of achievements in this area.

More recent work related to runtime migration of entire operating systems is presented in [2]. The most important advantages of this approach are: reduction of migration time to only several dozen milliseconds, and limiting the problem of residual dependencies between source and destination locations.[1] However, the important requirement is common network-attached storage (e.g. NFS) between the source and destination nodes which have to be parts of the same LAN. Moreover, the problem of residual dependencies remains a burden even when

[1] The residual dependencies problem involves the level of dependency of a migrating entity on the source host. It is the main factor which restrains broad use of migration mechanisms as it substantially reduces fault tolerance of the system.

local devices are considered e.g. when the target node does not provide a given device, present at the source.

Another approach to migration was undertaken at the level of OS processes. Significant research was performed in this area, resulting in several OS solutions, such as Sprite [3], Amoeba [4], RHODOS [5] and many others; however, only a few of them are used today [6]. Process migration is not available in modern, popular operating systems such as MS Windows, Linux and UNIXes. This is mainly due to the complexity of the mechanism and undesirable effects of state dispersal which directly result from the problem of residual dependencies.[2]

Yet another level where a migration facility may be introduced is the mobility of objects. Some languages and environments have been created with migration procedures in mind – e.g. Emerald [7] and COOL [8] but it is a far too complex a mechanism to be concealed underneath high-level language notation. Hence, more recent platforms such as CORBA, Java and .NET effectively implement migration [9] but the mechanism itself is not embedded in them.

The presented paper describes an approach to providing migration of CORBA Components which, in the context of growing interest in COA, may be considered interesting. Through balanced granularity — a component is "larger" than an object and usually "smaller" than a process — movement of components remains a flexible and efficient mechanism and may well support adaptation of application performance, which was the primary motivation behind the presented solution. The proposed extension of the CCM model ensures *weak mobility* (as defined in [10]) which is in contrast to *strong mobility*. The former is migration of "a code accompanied by some initialization data" — in this case the code and the state of a component, whereas the latter provides migration of the code and execution state which is generally more flexible but hardly possible to provide at middleware level.

In [10] there is presented a portable serialization mechanism which allows storing the state of a CORBA object and exchange it between different language domains. The mechanism described in this paper do not provide portability across languages, however offers a solution for migrating code of a component during its runtime with respect to the operations invoked on and by the component.

3 Migration Mechanism

Throughout this work the notion of migration is defined according to [11] as follows:

> object migration refers to the ability to move an object from one address space to another (change its physical location) without breaking references to that object currently held by clients.

This definition comes from a paper related to the CORBA platform, but the *object* mentioned above should not be perceived in OO categories. The notion

[2] Due to residual dependencies, multiple migration of entities results in the functioning of those entities being dependent on more and more systems.

may well refer to any code running in an environment, which is able to change its location. It is important to note that the presented definition, by stressing preservation of references between a migrating entity and its surroundings, forces the migration mechanism to ensure that following migration, communication with the entity shall progress as before.

The CORBA environment is well suited to resolving issues related to migration of objects. Mechanisms such as *Request Processing Policy*, `ServantLocator`s, *Servant Retention Policy*, `ForwardRequest` exception, etc. may well be used to support a migration facility. As shown later in this paper, all of them are also used to provide migration of components, hence in order to clarify how to move a component from one place to another, it is important to analyze how, in general, movement of objects may be performed.

There are several stages which a running object has to go through when migrating from one place to another:

1. **Suspending** the state of the object which is required to store its state consistently. The main issue here is that following suspending the CORBA platform still has to deal with incoming, ongoing and outgoing requests. Section 5.1 presents these problems in more detail.
2. **Storing** the state of the suspended object, alone or together with code. Which action is to be performed depends on the availability of the code at the destination. It is also crucial to answer the question of what state the object is in. If an object is connected with others, we must know whether they need to be copied as well or perhaps accessed remotely (shallow/deep copy problem). To make things even more complicated, storing state may also take into account heterogeneity of the environment and prepare a copy in an easily transferable format. Some of the issues mentioned here are covered in [12,13,14].
3. **Moving** the state between the source and target locations. This step is quite straightforward but in case of problems with transferring data, it should be possible to roll back the whole process and return the system to the state just before the suspension. Migration requires the target location to be ready to accept incoming objects, hence an appropriate infrastructure must be prepared at the destination.
4. **Loading** the state of the object at the destination. This step requires the code of the moved object to be available at the destination. In the case of heterogeneous environments, such as CORBA, this requirement is sometimes hard to fulfill — e.g. movement of a Java implementation of an object to an ORB for C++ language. Loading is much easier if we can assume platform homogeneity, such as that offered by Java or .Net environments.
5. **Reconnecting** of the moved entity in such a way that every other object (or, more generally, client) communicating with the migrating object should not see any change in behavior. There are three possible techniques of referencing a moved object: (1) deep update, (2) chain of reference, or (3) use of a home location agent. More details about this issue are presented in Sect. 5.3.

6. **Activating** the object at the new location followed by destroying it at the previous location. This is the final step which ends the whole process of migration and results in a fully functional system.

A crucial issue when considering migration is to shorten the time required to proceed through all the above stages, guaranteeing a more responsive and reliable mechanism. An important fact is that after suspension and before activation the object must not respond to any requests which can change its state. Otherwise, the stored state of the object would not match the actual state altered by the invocations and this would lead directly to loss of information.

4 Mobility with CORBA Component Model

The CORBA Component Model [15] defines an approach to designing, implementing and assembling component applications in the CORBA environment. By means of a new, extended version of IDL, it provides designers with an easy yet powerful way to define a component. Components may be equipped with several kinds of ports by which they are connected with other components or their execution environment. The model also introduces a new language, the Component Implementation Definition Language (CIDL), to describe implementation details of a component e.g. its lifecycle, persistence details, etc.

The CORBA Component Model does not in itself provide any mechanism which facilitates migration transparency i.e. movement of components between different locations. It is the goal of the presented work to describe steps which were taken to extend the CCM model and verify the extension on one of the available CCM implementations, namely OpenCCM [16].

A component in CCM may be perceived as having two sides:

– external side — visible to clients, defined by means of the IDL3 language which allows creating component definitions with attributes, ports and inheritance details. The basis for this part is the `CCMObject` interface,
– internal side — visible to a container, defined by means of the CIDL language. The basis for this part of a component is the `EnterpriseComponent` interface implemented by component executors.

The presented solution extends both sides of the component definition, allowing easy control of the migration mechanisms by an external entity. A prototype, called the *Component Migration Service* (CMS), has been developed for the purpose of evaluating the approach. This prototype is presented in Sect. 6.

4.1 External Interface

As mentioned earlier, migration consists of several stages: *suspending*, *storing*, *moving*, *loading*, *reconnecting*, and *activating*. In order to control movement of a component by an external entity it is necessary to extend the existing `CCMObject` interface with suitable operations. As shown in listing below, most of the steps described above have their counterparts in the proposed extension.

IDL definition of CCMRefugee interface, an extension to the original CCMObject interface

```
interface CCMRefugee : ::Components::CCMObject
{
    void refugee_passivate( );

    void refugee_activate( );

    void refugee_store( out Criteria the_criteria );

    void refugee_load( in Criteria the_criteria )
    raises( InvalidCriteria );

    void refugee_remove()
    raises( ::Components::RemoveFailure );
};
```

The meaning of the operations is consistent with the descriptions given in the previous section. The only two missing operations are `move` and `reconnect` which are included at the factory level (i.e. `CCMHome`). This decision is imposed by the fact that in order to move or reconnect a component it is necessary to destroy and create its instances, which is the primary goal of a factory.

4.2 Factory Involvement in Component Migration

In order to move a component to a new location, the destination must be prepared to accept the component. The presented solution does not introduce any special entities which carry out creation of a migrant at the destination. The CCM model provides a standard factory interface for every component, namely `CCMHome`, which may be simply extended to fulfill the requirements associated with accepting migrants. As shown in the following listing the `CCMRefuge` interface has four operations supporting movement of components.

Extensions to the factory interface are twofold:

– required at source location: `refugee_freeze`, `refugee_moved` and `refugee_unfreeze` operations. The aim of the first is to prepare a component and the infrastructure for movement. The second is responsible for reconnection of the moved component at the source location. The last extension is to be called in case of movement failure, when there is a need to reverse passivation of a component and return the system to the state from just before suspension,
– required at destination location: `refugee_accept` is invoked to ask the target to accept a migrating component. The operation returns a newly created incarnation of the component, used further by CMS to reconnect references. In case of problems, the operation throws an `InvalidCriteria` exception to signal the CMS to roll back the whole migration attempt.

IDL definition of CCMRefuge interface, an extension to CCMHome interface

```
interface CCMRefuge : ::Components::CCMHome
{
    Criteria refugee_freeze( in CCMRefugee refugee_here )
    raises (::Components::CCMException);

    CCMRefugee refugee_accept( in Criteria refugee_state )
    raises (InvalidCriteria);

    void refugee_moved(
        in CCMRefugee refugee_here, in CCMRefugee refugee_there);

    void refugee_unfreeze( in CCMRefugee refugee_here )
    raises (::Components::CCMException);
};
```

The presented enhancements are used by CMS as depicted in Fig. 1. From the point of view of CMS, migration consists of three basic stages: (1) freezing the state of the component, (2) moving the component to the target location, and (3) reconnecting the component at the target location.

At any stage following passivation of a component, a failure may occur. In such a case migration shall be immediately abandoned and the system shall be restored to its original state as fast as possible. Then, instead of reconnecting by means of `refugee_moved` it is necessary to invoke the `refugee_unfreeze` operation which reactivates the component at the source location.

4.3 Extended Component Lifecycle

Apart from the extensions which enable migration control, some enhancements are required at the internal side of the component, namely its executors. They provide a way to inform the programmer about component state changes.

The operations included in `RefugeeComponent`, which is a basis for executors of movable components, reflect directly operations published in `CCMRefugee`. This is because `CCMRefugee` delegates requests to the appropriate executor, implementing the `RefugeeComponent` interface. Operations of this interface indicate changes in components' lifecycle and should be used by a component developer to control resource usage, progress in communication, internal state of the component, etc. For this reason it seems worthwhile to describe the exact meaning and proposed use of each operation:

– `ccm_refugee_passivate` — is called just before passivation of the component. The developer shall use this indicator to prepare the component for the storing phase, i.e. the component should interrupt any activities which may change its state during migration. As discussed later in Sect. 5.1, the range of activities which the developer should perform during this call depends on the way the component is implemented,

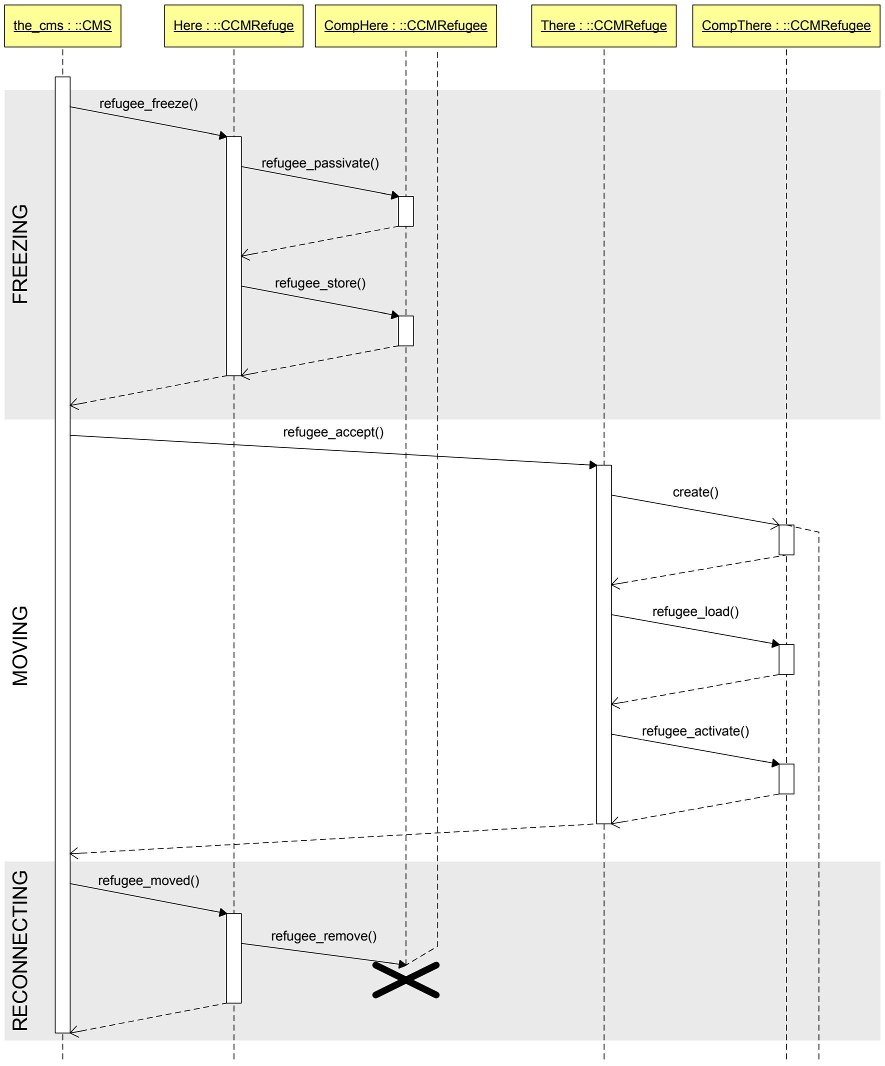

Fig. 1. Sequence diagram of successful migration between locations labeled HERE and THERE

– `ccm_refugee_store` — is called to store the state of the component. Although some languages, such as Java and .Net, can serialize classes automatically by means of the reflection mechanism, in this work a manual approach is adopted in order to preserve greater portability of the CORBA environment,

- `ccm_refugee_load` — is opposite to `ccm_refugee_store` and shall be used by developers to restore the state of the component. Once this operation is invoked, it is certain that the component is located on the destination host.
- `ccm_refugee_activate` — may be called in two cases. Firstly, following successful migration the operation is called on the newly created component at the target location to indicate that the component is going to be activated and has to be ready to resume work. Otherwise, when migration fails, `ccm_refugee_activate` is called at the source location to indicate that the component returns to its normal operation.
- `ccm_refugee_remove` — is called on the component at the source location whenever the migration attempt is successful. The aim of this operation is to indicate that the component should release all resources acquired during its work at the source host.

5 Migration Internals

The interfaces presented above provide a convenient way to control migration of components. However, in order to successfully move a component it is necessary to consider some crucial issues such as processing of requests, reconnection and resource usage. The last issue is particularly important as it imposes some constraints on how resources can be used by a mobile component. The following sections provide a brief discussion about these problems.

5.1 Dealing with Requests on Suspension

As far as migration is concerned, one of the major issues is dealing with requests which an object is or should be involved in. This problem arises when the object is going to be suspended in order to preserve the consistency of its state, but it is still entangled in some operation. In general, three possible cases are relevant here: (1) *incoming requests* invoked on the object during suspension state, (2) *outgoing requests* invoked by the object before suspension, and (3) *ongoing requests* invoked on the object before suspension.

The solution to the first case is to collect all incoming requests until the object is again reactivated. In the case of successful migration, all these requests are redirected to a new location using the CORBA `ForwardRequest` exception.

The second case is more troublesome. It is important for the passivation of the object to be performed if all outgoing invocations are already dealt with. Otherwise, the returning result could introduce some inconsistencies between the stored and real state of the object. In order to deal with such cases automatically, Container Portable Interceptors (COPI) are required. Unfortunately, the COPI specification has only been adopted recently and it is yet not widely implemented by CCM platforms. Without COPI there is no easy way to determine the number of outgoing requests on the middleware level. In the proposed extension the solution to this issue is left to developers who need to be aware of all outgoing requests whenever the `ccm_passivate` operation is called.

Container Portable Interceptors may also be a very convenient and elegant way to deal with the third problem i.e. ongoing requests. In this case, however, their functionality may be easily overtaken by a `ServantLocator`. Two operations of the servant manager — `preinvoke` and `postinvoke` — are used to count the number of ongoing operations. The locator ensures that passivation does not occur until all the operations are finished and, by collecting all incoming requests, guards the object from being bothered. Unfortunately, such a simple solution may sometimes impose substantial delays in suspending a component and developers should take that into account.

An important fact is that the solutions proposed above do not protect the component from all state consistency-related problems. For example, if the component interacts with the environment by means other than CORBA, there is no easy way to provide a general solution at the level of the CCM container.

5.2 Constraints on Resource Usage

As mentioned above, whenever a CCM component communicates with the environment by means other than CORBA it may create problems with state and communication consistency. The very same problem occurs when dealing with a local filesystem, local devices, threads running on a source host and all other local resources which are not accessible in the address space of the destination host.

Nevertheless, in order to give developers substantial freedom of using software and hardware platforms for component hosting it is not desirable to limit access to local resources or native communication technologies. Instead, the lifecycle of a component has been extended, providing programmers with means to be aware of oncoming migration. There are two important cases to be considered: (1) departure from a source host, and (2) arrival at a destination host.

Successful departure is signaled by two operations: `ccm_refugee_passivate`, and `ccm_refugee_remove`. Passivation means that a component should cease all activities which might change its state. Obviously, this may have an important impact on communication, thread usage and sometimes resource allocation. The second operation indicates the moment to free all gained resources, destroy all local allocations, etc. This operation means that the component has been effectively transferred to a new location and may be completely destroyed at the source.

Signaling component arrival at a destination host is done with the use of the `ccm_refugee_activate` operation which should have semantics similar to both the `configuration_complete` and `ccm_ activate` operations originally called by a CCM platform when the component was instantiated.

5.3 Reconnection

Another very important problem related to migration is reconnection between the migrated component and all other clients, objects and components which it interacts with. As mentioned earlier, there are three possible techniques of resolving this issue: (1) deep update, (2) chain of reference, and (3) use of a home location agent.

The first technique requires all clients of the component to update their references following migration. This is very expensive approach and, in fact, not a viable one in distributed environments such as CORBA, since the clients may not yet exist when migration occurs [11]. The second technique assumes that after movement the component leaves a trace at the previous location which points to its new incarnation. From the point of view of a client, this is much more convenient, however, each movement makes the chain longer and longer, eventually introducing significant inefficiencies in communication and being more prone to failures (vide residual dependencies).

The last approach seems to be the most appropriate for resolving the problem of referencing. On the one hand, clients can refer to the moving component through a persistent reference of its home location agent. On the other hand, the home agent is the only entity that should be informed about location changes. This guarantees that consecutive migrations of the component do not incur any additional delays in request processing.

The use of the home location agent has its drawbacks. Firstly, it also introduces the problem of residual dependencies, although to a far lesser extent than the chain-of-reference method. Secondly, use of a separate home location object introduces additional costs even when the movable object is managed by the same object adapter as the home agent. To reduce this overhead a `ServantLocator` extension is proposed, which, by maintaining an additional *migratory table*, becomes the home agent itself. The `ServantLocator`'s `preinvoke` operation is responsible for searching through the migratory table and returning a `ForwardException` if the received request is directed to a component which has migrated away.

6 Evaluation of Efficiency

Despite all the potential advantages stemming from introduction of a migration mechanism to the CCM platform, it is not surprising that its use incurs additional delays on processing requests which, in consequence, lowers the overall throughput of an application. Irrespective of how efficiently migration is performed, the source of the loss of efficiency is at least twofold.

Firstly, it is connected directly with the means by which requests are processed. Clients which use a reference of a moved component have to submit requests twice: first to the home location to get the current reference of the component, and then again, using the acquired reference, to the component itself. This overhead is usually substantially reduced by an ORB which caches references returned in the first step, ensuring that all consecutive requests are sent directly to the new location. Nevertheless, for the first invocation following migration the overhead still persists.

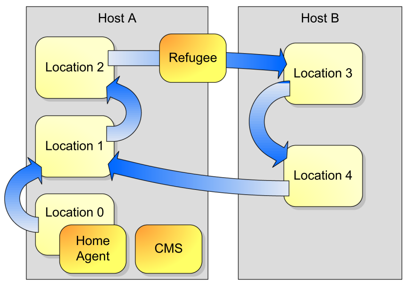

Fig. 2. Migration of a refugee in a testbed used to evaluate overhead of the migration mechanism

The other and more severe reason for loss of efficiency is the time required to move a component between two locations. In order to perform migration, the component is suspended for the duration required to transfer its state. Obviously, the longer this interval is, the less requests the component is able to process. That is the main reason why optimization of this step is a crucial part of providing a mechanism which would offer acceptable responsiveness of migrating components.

Figure 2 presents the testbed used to evaluate this kind of overhead. There were five `Refuge` locations placed in two hosts, A and B, connected with a 100 Mb/s LAN network. Location 0 hosted the Home Agent of a moving component which migrated between locations 1–4.

The testbed was used to evaluate migration of different kinds of components. Table 1 lists the duration required by migration between the locations in relation to the complexity of the component.

Table 1. Time [ms] required to perform migration in relation to complexity of the component

	$L1 \rightarrow L2$	$L2 \rightarrow L3$	$L3 \rightarrow L4$	$L4 \rightarrow L1$
No ports, no data	**109**	**142**	**183**	**147**
One facet	125	157	194	160
One receptacle	108	143	188	151
Some data	114	145	186	151
Five facets	127	168	222	175

Basing on the data presented in the table, it is worth to point out two interesting facts. First, moving a component with a facet or event sink consumes more time than moving a component with only a receptacle or event source. This is because facets and event sinks are CORBA objects and have to be stored together with the state of the migrant in order to reconnect it properly. Second, as can be seen, migration between locations 3 and 4 yielded the worst results, whereas movement between locations 1 and 2 proved fastest. The reason for this is that, originally, the components were placed on host A at location 0, hence local updates of the Home Agent from location 1 and 2 were faster than network communication between locations 3 and 4 and the agent. Additionally, the location of CMS, which was placed on host A, was also important. This, again, resulted in better performance if migration involved locations 1 and 2.

The results collected in the table convey important information. They provide an order-of-magnitude assessment of the time consumed by component migration. The most important case is the one when a component does not have any ports and data. It shows pure migration overhead while other results are distorted by serialization and transfer of code over the network. The results should also be taken into consideration to estimate the number of operations per second which the moving component is able to perform reliably. However, the exact performance of the component highly depends on many factors such as the length of the ORB's request queue and the implementation of lifecycle operations described in Sect. 4.3.

7 Conclusions and Future Work

The presented work describes extension of the CORBA Component Model with a migration facility. The adopted approach does not provide a fully transparent solution which, due to the important problem of residual dependencies, seems to be unattainable. Instead, we propose an extension of component lifecycle, providing programmers with an interface to deal with migration in a proper way. This is consistent with the approach proposed by the original CCM model where a component is notified about configuration completion, activation, removal, etc. Moreover, making programmers aware of component mobility does not impose substantial constraints on the range of resources and communication technologies used. The cost is that the programmer is responsible for manual preparation of a component for a migration attempt. However, at the level of middleware, it seems hard — if indeed possible — to automatically generate the whole required migration infrastructure for a component. The container is responsible for CORBA communication only, and any other technologies are out of its scope.

This situation would improve if the CCM platform implemented the *Streams for CCM* specification [17]. Local resources could then be accessed by means of sink and source ports, allowing for better detachment a component from its execution environment and, in consequence, more transparent migration. This area is a potential direction for further research.

The proposed migration mechanism is a prototype working with `session` components only. It is necessary to develop and test a mechanism suitable for `process` and `entity` component categories, as well as components with multiple segments. Unfortunately, OpenCCM, the platform used as the development environment, does not support components of other types than session, hence this direction of work is currently hampered.

Other possible development directions are related to integration of the migration mechanism with CORBA services, especially Persistent State Service and Transaction Service, as well as better integration with OpenCCM code generation tools. Nonetheless, the mechanism presented in this paper allows for further work concerning adaptive deployment and execution of applications. The migration facility, as one of the executive mechanisms, plays there an important role giving an Adaptation Manager a chance to control component arrangement of an application.

References

1. Milojičić, D., Douglis, F., Paindaveine, Y., Wheeler, R., Zhou, S.: Process migration. In: ACM Computing Surveys, pp. 241–299 (2000)
2. Clark, C., Fraser, K., Hand, S.: Live migration of virtual machines. In: Proceedings of 2nd Symposium on Networked Systems Design and Implementation (2005)
3. Douglis, F.: Transparent Process Migration in the Sprite Operating System. PhD thesis, University of California at Berkeley (1990)
4. Mullender, S., van Rossum, G., Tanenbaum, A.: Amoeba: A distributed operating system for the 1990s. IEEE Computer 23(5), 44–53 (1990)
5. de Paoli, D., Goscinski, A.: The RHODOS migration facility. The. Journal of Systems and Software 40(1), 51 (1998)
6. (openMosix project) Web site at `http://openmosix.sourceforge.net`
7. Hutchinson, N., Raj, R., Black, A., Levy, H., Jul, E.: The Emerald programming language. Technical report, Institution (1987)
8. Habert, S., Mosseri, L., Abrossimov, V.: COOL: Kernel support for object-oriented environments. In: Meyrowitz, N. (ed.) Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), pp. 269–277. ACM Press, New York (1990)
9. Tröger, P., Polze, A.: Object and process migration in.NET. In: Proceedings of the Eighth International Workshop on Object-oriented Real-time Dependable Systems, pp. 139–146 (2003)
10. Fuggeta, A., Picco, G., Vigna, G.: Understanding code mobility. IEEE Transactions on Software Engineering 5, 342–361 (1998)
11. Henning, M.: Binding, migration, and scalability in CORBA. Communications of the ACM 41(10), 62–71 (1998)
12. Killijian, M.O., Ruiz-Garcia, J.C., Fabre, J.C.: Portable serialization of CORBA objects: a reflective approach. In: OOPSLA, Seattle, USA, pp. 68–82 (2002)
13. Object Management Group, I.: Externalization Service Specification. Object Management Group, Inc. Version 1.0 (2000)
14. Object Management Group, I.: Life Cycle Service Specification. Object Management Group, Inc. Version 1.2 (2002)
15. Object Management Group, I.: CORBA Components. Object Management Group, Inc. Version 3.0 (2002)
16. (OpenCCM — the open CORBA components model platform) Web site at `http://openccm.objectweb.org`
17. Object Management Group, I.: Streams for CCM. Object Management Group, Inc. Draft Adopted Specification (2002)