Object and Process Migration in .NET

Peter Tröger and Andreas Polze
Operating Systems and Middleware Chair, Hasso–Plattner–Institute at University of Potsdam, Germany
{troeger|polze}@hpi.uni-potsdam.de

Abstract

Most of today's distributed computing systems in the field do not support the migration of execution entities among computing nodes during runtime. The relatively static association between units of processing and computing nodes makes it difficult to implement fault-tolerant behavior or load-balancing schemes. The concept of code migration may provide a solution to the problems mentioned above. It can be defined as the movement of process, object or component instances from one computing node to another during system runtime in a distributed environment.

Within our paper we describe the integration of a migration facility with the help of Aspect-Oriented Programming (AOP) into the .NET framework. AOP is interesting as it addresses non-functional system properties on the middleware level, without the need to manipulate lower system layers like the operating system itself.

We have implemented two proof-of-concept applications, namely a migrating web server as well as a migrating file version checker application. The paper contains an experimental evaluation of the performance impact of object migration in context of those two applications.[1]

1. Motivation and Introduction

Most of the distributed systems in the commercial area do not allow the re-binding of an execution entity to a different node during runtime. With every entity bound to one node these systems have typical problems such as 'single point of failure' or a missing overload protection.

Object and process migration is a possible solution to the problem mentioned above. Most of the migration facilities described in literature are implemented on the level of processes [9]. There exists a vast amount of studies and research work in this area, which concentrates on process migration facilities inside distributed operating systems. Additional research focuses on integration of migration concepts into object oriented languages and systems [2]. There are several practical reasons to integrate migration in a distributed environment: Load balancing, load sharing, application concurrency, object persistence, efficient remote procedure calls / resource access [10] or the pervasive computing approach [5].

Our research focuses on a migration facility for active and passive objects in the .NET framework. Our work follows the idea of Aspect-Oriented Programming (AOP), which allows us to address non-functional system properties on the middleware level, without the need to manipulate lower system layers like the operating system itself. Most of the features required for the implementation of object or process migration (such as location transparency or a machine-independent executable format) are already present in the .NET frameworks, so the integration of such a concept is a natural extension of the system capabilities.

The remainder of the paper is structured as follows: Section 2 gives an overview over related work. Our approach to language-independent object migration in a component framework is presented in Section 3. Section 4 discusses implementation issues, whereas Section 5 presents an experimental evaluation of our approach based on two proof-of-concept applications. Section 6 finally concludes the paper.

2. Related Work

In his classification approach for mobile code architectures Picco [7] makes a fundamental distinction between *weak mobility* and *strong mobility*. In systems with weak mobility the migrant is either a data object having no own path of execution or an executable object, which starts execution from the beginning after migration. The second group of entities are strong mobile objects, which are interrupted in their work for the migration and carry forward the execution on the destination node. Executed objects can be further divided into interpreted code objects and native code objects.

For weak mobility with data objects the persistent storage concepts in most commercial component frameworks are a good example. Executable code that is started from the beginning after the transfer is also widely known

[1] This work has been sponsored by Microsoft Research Cambridge under agreement number 2001-61.

mechanisms also offers the possibility for performing a platform-independent migration. If a chosen component framework is available on multiple platforms, the abstraction approach guarantees that serialization and de-serialization work also while crossing platform boundaries. The migration framework presented here has been implemented on the commercial .NET implementation under Windows 2000; however, it has been ported and is functional under the shared source implementation of .NET (Rotor) under the Windows XP, Free BSD and MacOS operating systems as well.

In order to be used with our framework, all migrants (classes that shall be serialized) must be marked with the [*Serializable*] attribute. In the case of active objects, the instruction pointer for the current point of execution is a relevant property, which is not saved with the serialization functionality. We have solved this problem by marking a special *re-entry*-method within the migration aspect code, which is called automatically after restoration of a migrant on its destination. Serialization occurs in the class scope. Therefore, all relevant state information must be encapsulated in the form of class members. Data residing outside of a migrant's scope is not covered by the serialization mechanism. We have implemented a separate *post-migration*-handler, which takes care of these global data.

Transfer of Code and State Information

After the successful identification of a destination and the storage of the migrants' information all relevant data must be transferred to the new host. The architecture introduced in this paper contains a migration server that is available on every host. In addition to the task of answering requests for migration destinations, this module also works as receiver for a migration data stream. It is also responsible for the continuation of the migrant. The relevant subtopics in the handling of state and code transfer are completely managed by this instance.

One possible simple improvement in a component framework is the caching of already transferred binaries inside of the migration server. On later re-migration events of the same entity the framework is able to reuse the already transferred binary code. The .NET framework has a mechanism of holding shared assemblies in a global cache (GAC) that could be utilized to support migration. Our current implementation described here uses a mechanism where the migration facility on the source node asks the chosen destination host for the availability of the migrant code base. The unique identification of migrant assemblies relies on the typing mechanisms of the component framework.

Handling of Residual Dependencies

In all migration frameworks, regardless of their scope, there is always the problem of local references that cannot be easily transferred to the new host. These local references or dependencies mostly refer to system resources that are managed inside of the operating system - file handles, network sockets, shared memory regions or other location dependent information. Another problem is the *accessibility problem* that occurs always in such environments. If a running entity accessible for other entities in the distributed system is moved, than there must be a mechanism that ensures that the migrant remains accessible even if it is at the new host.

With the goal of non-intrusiveness and the flexibility of AOP it seems to be more practical to give responsibility for non-migratable resources and location transparency not to the framework but to a use case specific module. This module can be dynamically connected to the migrant through aspect mechanisms. Our approach allows for implementation of generic handlers, which use classical forwarding or remote access solutions. It is also possible to introduce specific handlers (so-classed *post migration handlers* – PMH) that match exactly the use case and communication characteristics of a migrated application. The PMH is responsible for the complete handling of resources that cannot be migrated by the framework itself. The migration framework is responsible for giving all available information about the problematic resources to the PMH. Additionally it must give the PMH a chance to survive as active task at the source node even if the migrant has left. However, one has to take care of the fact that the migrant could go back to its original host. In this case there must be a defined way for the active PMH handler to finish its work in a way that the migrant application is able to continue execution without interfering the forwarding mechanisms.

Another accessibility-related problem are blackout-periods, which occur when an application migrates. Remote calls arriving during migration cannot be handled immediately. However, standard retransmission and flow control mechanisms of the widely used TCP transport protocol are a satisfactory solution. This assumes that the blackout-period of an application is short enough to not go beyond the timeout value of the protocol.

Continuation of Execution

A transferred active entity must be restored from the state information and must be continued in its execution on the new host.

In our approach, we chose to create a semi-transparent solution with the help of AOP. The idea here is to declare the re-invocation method through the aspect code that is interwoven with the migrant. This can be seen as an acceptable solution because it is already clear that the aspect code designer must have knowledge of the internal functionality of the migrated application. With the declaration of a re-entry method the framework can easily perform the restart operation on the destination host. If

since many years, e.g. Unix remote shell (rsh), the Java RMI or Microsoft ActiveX. Concerning the type of the executable object, systems with interpreted executables have some advantages over those with native code execution. This factor is especially important if the migration facility shall run in a heterogeneous environment [6].

There are several practical subtopics in the area of code migration that differentiate related work: Kernel related data stored in the address space of the process (for example file handles) become a relevant problem during the state preservation and restoration [3]. Sprite [9] solves this problem with the remote usage of location depended resources on the home node.

There are also several approaches in the different practical systems to receive the relevant data directly from the execution environment on the source node. Condor [6] relies on wrapped system libraries that allow the continuous supervision and logging of created and released system resources. In combination with a modified core dump functionality for considering dynamic data this solution works without modifications to the underneath standard UNIX kernel. In a component framework the component connector concept could be used similarly to introspect the usage of resources transparently. Also the component container architecture can be used for such introspections.

Some systems work with a complete virtual machine on top of the operating system. This strategy is mostly used in agent systems in which a special layer rests on top of a normal UNIX system [1]. A modified Java virtual machine is also being used to support strong code migration [11]. Agent systems additionally use a specialized compiler to include necessary information (such as code preemption points) in the executable. Most language extension based solutions are marking migration relevant data in the code, which allows the compiler to inject the needed migration relevant instructions at the right point. An example for this class of systems is the SOS system [13], were C++ classes can be marked as dynamic and the compiler inserts an indirection table for pointers. With this method the indirection table can be adjusted on the destination machine for the new address space.

The TUI system [14] uses debug information collected at compilation time and generates intermediate data for heterogeneous migration purposes. Ferrari [6] enhanced this idea by suggesting a generation of intermediate code that is able to restore the execution state on the destination machine. The Java Tube system [8] breaks the program into scalar fragments at compilation time and saves their overall state after the execution of one fragment. The author calls this *high order state saving* because the state is saved at the virtual machine level.

However, several architectures for strong code mobility make the migrant directly responsible for state preservation. One very common solution in this respect is the specification of a compulsory management interface that is called before migration start and after migration end. This approach is used in the ANSAWare migration extension Zenith [4] and in the SUN JINI system. The SOS system [13] calls a re-initialization constructor on the object after the migration is completed.

3. Migration within Component Frameworks

The term *migration* is used in several contexts in practice. Figure 1 shows a possible classification of different approaches to migration.

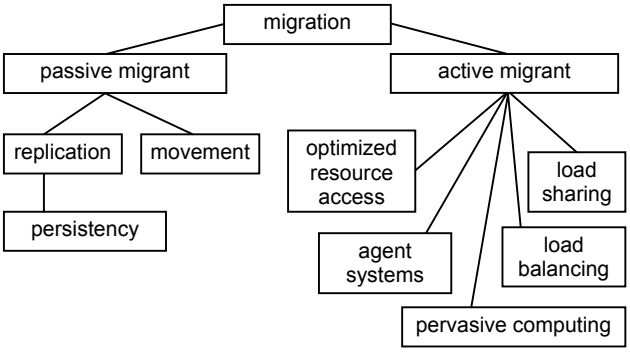

Figure 1: Classification of migration types

Basically there is a distinction between the migration of passive and of active objects. Passive objects do not have an own path of execution. Examples for this are data objects or instances of library classes. A passive object can be replicated or moved to another host. Modern component frameworks offer a replication mechanism under the term *serialization*. It allows saving the actual state of an object to a persistent store. Since the original object can be further used after its serialization, the mechanism is classified here as *passive object replication* operation. In case of an object movement the object is no longer available at its source host, which leads to the classification as *passive object movement* operation. Both variants have to consider the consistency problem for replicated data.

The migration of active objects deals with independently running software modules or executables. This can affect interpreted or compiled processes or objects. Within this paper the term *migration* is used for a movement operation of an actively executed object. The replication of an executed active object cannot be seen as migration activity. This is reasoned by the fact that no consistency model for its operations can be guaranteed or modeled here, while for a movement a strong consistency is always aimed through several standard mechanisms.

Our approach to object migration in component-based frameworks has to consider a number of fundamental design decisions, which are discussed below:

Migration Decision

In order to support binary reuse of components, migration policies should be dynamically attached to possible candidates for component migration. The main idea here is to build a framework where objects residing inside a component can perform self-initiated as well as externally triggered migration. This goal is reached through the extension of the migrant code with mechanisms of aspect-oriented programming (AOP): The aspect code for a migrant checks the policy at dedicated points of execution. If the policy claims a situation where the migration should happen than the aspect code has to perform the search for a matching destination. After a successful search the aspect code can initiate the transfer to a new host. Alternatively, the aspect code may trigger a migration explicitly without taking care of a policy rule and only with regard to the internal state of the migrant.

In our approach, the aspect code, which is interwoven with the migrant, calls the check routine of a so-called policy module at specified points of the execution. The policy module starts the migration with library functions if the policy condition is met. The aspect code is also able to start the migration directly because of a special condition. This happens if it detects a specific internal state of the migrant. The state can depend on member variables or function results.

The other relevant part of the architecture, the migration server, is needed to locate a destination prior to a migration step. If the migrant (or more exactly its aspect code) does not name an explicit migration destination, the library asks around in the network for a matching destination. In the actual design this is simply done by sending the policy type as multicast network message. All migration servers in the multicast group check if the policy module is available locally. In the positive case they answer if their own policy check allows a new migrant. In the negative case no answer is send. This technique for location of a destination allows parallel work of different types of migrants inside the same distributed system. The source node takes the first positive answer and initializes the migration to the host where the message was coming from.

Preemption of the Application

After the selection of a migration destination the next step is the safe preemption of the application. It must be ensured that both the system and the migrant itself are not left in an inconsistent state. The system must be able to continue its general work after removing the migrating application. The migrant should be in an execution phase where it is possible to save and later restore its internal state completely. In all former solutions this part of the whole procedure leads to some non-trivial problems. One example is the migration of processes currently performing a pending system call. If the underneath system is not prepared for the possibility that a program is removed in this state an instable system environment could arise.

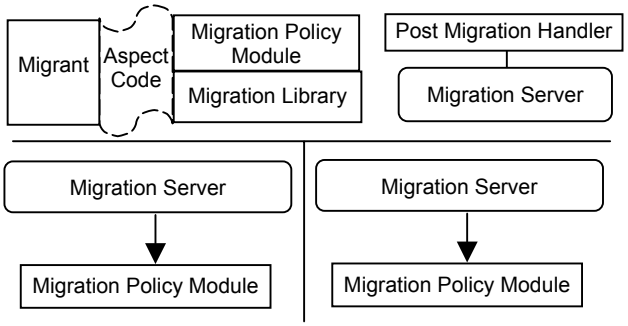

Figure 2: Concept for a flexible migration framework

Our approach again utilizes the flexibility of AOP. Assuming a message-driven, cyclic execution model, the aspect-weaver identifies possible migration points at the end of each message-handling function. This approach ensures that the migrant is always in a save state for transfer. There is no possibility that pending system calls or other problematic actions are performed during this time. The assumption a cyclic execution model somewhat restricts applicability of our current solution. However, this is not a problem with passive (server-type) applications (which act on incoming requests – method calls), but rather in actively executing applications. Here it could be possible that major parts of the program rest inside the *main()* function without ever completing an execution cycle. In this case, which can be seen as well as bad software design style, the migration facility and the aspect code do not get a chance to check for a policy condition or to start a migration operation.

State Saving

After interrupting the migrating application, the next critical step is to save the current state of the object instances residing in the migrating components. Most modern component frameworks provide mechanisms for object serialization. Serialization is the process of saving an object state to a fixed storage. The advantage here is that the complete state inspection is handled directly on the level of the runtime objects. The framework is responsible for the platform-independent encoding of the state information (with respect to byte order or alignment problems), the recursive analysis of cascaded data structures and the proper saving of the correlation between data type, name and value. The usage of pure framework

such a method is not marked by the aspect code, than the framework could call the last method that was executed before the migration. This variant has to be used carefully to avoid state inconsistencies of the migrant.

The next chapter explains how the concrete implementation of the concepts presented here is accomplished within the .NET framework.

4. Implementation Issues

The whole framework was developed using the C# programming language. Figure 3 shows the general architecture of our migration framework.

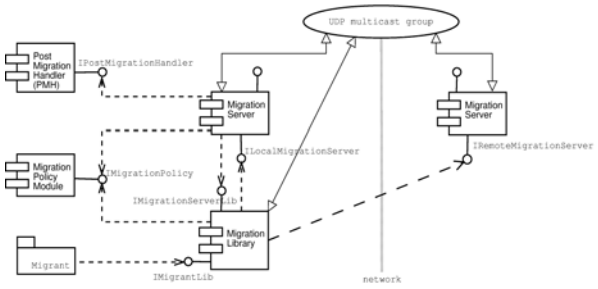

Figure 3: Architecture of the migration framework

The migration library is the functional heart of the framework. All primary functionalities for the migration are concentrated here. Each migrant application is bound to a library instance through its aspect code. The library offers the *IMigrantLib* interface for all the necessary functionalities.

The first action of the migrant related aspect code is the registration of the migrant itself, the regarding policy module and the post migration handler (PMH) module:

```
bool Start(object mig,
           string policyFileName,
           string pmhFileName);
```

The last two parameters are optional, so it is also possible to use the framework without a policy module or the post migration handler. The next parts of the *IMigrantLib* interface are the functions to start an explicit or semi-transparent migration attempt. This it is accomplished by three functions:

```
bool MigrateToHost(string destHost);
bool MigrateIfNeededToHost(string dest);
bool MigrateIfNeededSearchHost();
```

The first function starts directly a migration to the chosen host without checking any policy condition. With a call to the second function, the destination host is given but the decision if the migration is necessary is left to the policy module. The third variant leaves all decisions to the policy module and the migration framework. In this case the library is not only responsible for the local check up of the policy condition but also for asking for a matching migration destination within the network. This is achieved directly through a connection to the migration server UDP multicast group. The other functions of this interface are necessary for the concrete state saving and restoring in the context of the serialization procedure.

The second interface *IMigrationServerLib* offers all relevant core functionalities for the migration server. This mainly bears the subtasks during the restoring of the migrant in mind. Concretely the following functions are available:

```
MigrantEnv PrepareMigrantEnv(Stream stream,
                             int minStayTime);
bool IsAssembly(string assName);
bool RegisterPolicy(string aName);
void StartServerComm(string udpAddress,
                     int udpPort,
                     Type theServerType);
```

The first function creates a new execution environment for the migrant. In terms of .NET this yields to the creation of a new application domain and the de-serialization of the migrant in this new domain. Since the migrant is received as binary stream by the new host, this stream is given as argument to the function. The second argument is the amount of seconds the migrant has to stay minimally at his new host. The main intention for this factor is the avoidance of *'migrant flooding*': If for example a migration policy checks for the CPU load of the local machine then it could happen in a network wide overload situation that most of the machines are above the fixed threshold of the policy. This would lead to a continuous migration of all entities.

The result of the function call is an object that acts as handle for the new environment. The migration server manages these handles.

The server uses the second function *IsAssembly()* to check if a code assembly is already loaded when a remote node asks on the *IRemoteMigrationServer* interface for that assembly.

The last two functions are used at initialization time of the migration server. Firstly the policy modules that will be used for remote migration destination requests are registered with the *RegisterPolicy*() function. The *StartServerComm*() routine connects to the UDP multicast group of servers and establishes the remoting channel for the accessibility of the *IRemoteMigrationServer* interface.

During the implementation phase some general conceptual problems with the implementation and specific problems with the .NET framework appeared:

The .NET *Main()* function that is called first in an executable is marked as static function. Inside this function, a serialization cannot be started because there is no object instance available at this point. The migration cannot be started until the control flow moved into a user-created application object occurred.

A similar problem exists if the constructor of the application object performs the main work of the application. The constructor of the aspect proxy class is always executed after the base class constructor is finished. If the base class constructor calls virtual functions that are overridden by the proxy class it could happen that these methods call migration library functions before it is was initialized in the derived constructor method. This problem can be avoided through a proper handling in the aspect code.

The concept of using the serialization functionality for state saving leads to some limitations. The state of external assemblies or objects being used by the migrant is not saved. Also some deep-level data structures like the hash table collection are not completely savable. At the moment there are no viable solutions for this kind of problem available. The post migration handler modules could solve part of the deep copy problem. Saving external states of related assemblies can only be implemented in an intrusive fashion. We are currently inspecting the shared source implementation (Rotor) of .NET to explore solutions to this problem.

Modern applications usually work with multithreading. In .NET a threading functionality is also available. If the migrant application uses multiple threads several problems may arise. Thread local data in .NET is not saved during a serialization. Checking for termination of all threads contained in an application domain is another problem, which has to be solved to decide when a migrating application may migrate to a previously used location.

AOP under .NET

We have used *LOOM.NET* by Wolfgang Schult [12] to interweave the migration aspect code with an already compiled .NET assembly. Aspect code can be defined separately for the various programming language-entities: namespace, class, constructor, method or field. These concrete aspect implementations are grouped together and can be attached as package to a binary assembly. During aspect weaving, the *LOOM.NET* tool creates automatically a derived class from the original .NET class contained in an assembly. This proxy class contains the aspect code and can be compiled and linked to produce an extended version of the original assembly. The weaving mechanism is extensible by the aspect designer through the implementation of so-called extension modules.

5. Experimental Evaluation

We have implemented two proof-of-concept applications that utilize our migration framework. There are two different motivations for using migration in context of those applications: the migrating File Version Checker accesses data residing on local disks on different nodes in a network. It migrates to access large amounts of data locally. Our experiments show that there is a tradeoff between migration overhead and performance gains during disk access.

The web server uses migration for a different reason. It accesses data stored on a network file system and migrates itself to a different node when it encounters a dramatic increase of computational load on its home node. We could demonstrate that the performance gains from moving to a lighter loaded node outweighs the overhead required for forwarding http-communication between home node and current location of the web server.

Test Environment

All experiments were done on four machines of the same type (Pentium II 450 MHz, 128 MB RAM, Windows 2000, 100Mbit network). The assembly cache of our migration framework was modified in such a way that it always claims the non-availability of the requested assembly. The reason for this modification is a bug in the .NET framework. Also the security context mechanisms were switched off in the experiments.

The clocks of the 4 machines were synchronized through NTP over the Internet. NTP adjusts clocks with a drift of less than one millisecond. With the synchronized clocks and the time stamp logs from the 4 nodes it was possible to get a complete timing description of the several actions in the migration framework.

Experiments with File Version Checker

The first proof-of-concept application is a program to check file versions. It goes recursively through the directory structure from a given path and collects all files that match the search condition. For every file the version information is extracted. This is done for a set of given hosts. The result is a file that describes the differences between the hosts. The program can be used for automatic checks before software installations.

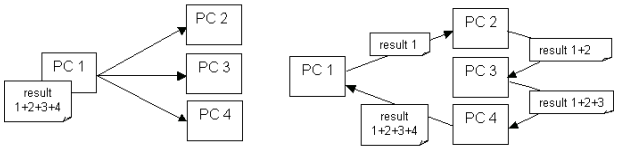

Figure 4: Non-migrating and migrating File Version Checker

The first version of this application is performing the same recursive check routine locally and than remotely on the other systems. The results of all these runs are correlated and written in the result file. The remote check is carried out through a network file system access on the administrative share of the remote host.

The migrating variant of File Version Checker starts also with the recursive check of the local resources. At the end of this function call the aspect code triggers an explicit migration to the first remote host. At this host the recursive check routine is called as re-entry method, so again a local file check is initiated. This procedure is carried out until the migrant returns to its home node where the result file is created.

The file version checker application can be seen as example for the 'migrate to resource' approach. The primary goal is the improvement of the execution time. The characteristic of the application reminds of the agent approaches in system management applications. With this working proof-of-concept scenario in mind several other agent-oriented applications are imaginable.

The tests were done with an increasing number of files to be evaluated. The aimed goal of the experiment was to show that through the advantage of accessing all files locally in every case the migrating solution should perform faster than the non-migrating solution.

In Figure 5 the application's runtime in relation to the number of files checked is shown. It can be seen in the graph that the version without migration performed faster than the solution with migration. This is due to the migration overhead, which correlates to the state information stored in the file version checker and increases with an increasing number of files to be checked.

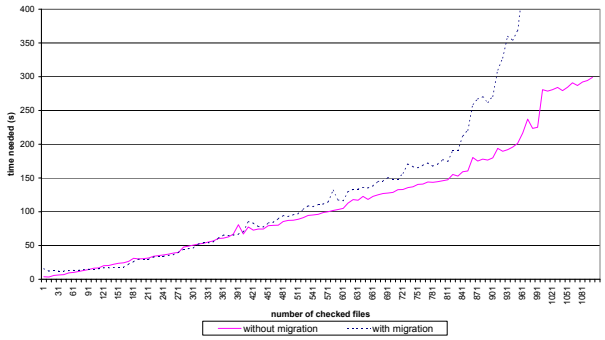

Figure 5: Performance comparison for File Version Checker

Another argument for the better performance of the non-migrating solution can be seen in file caching effects. All runs of the experiment were performed sequentially with a simple shell script. In this scenario the non-migrated application has the advantage of being speeded up by the operating system file access caching. The same files are accessed again and again from the same machine in the experiments. After an initial phase most accesses are answered directly by the local file system cache. The migrant application is restarted at every node. This leads to a much lesser effect of caching strategies and eats up the advantage of the local file parsing.

Experiments with a Web Server

Our second proof-of-concept application is a small web server. The base program was taken from a C# developer page[2]. The program is a simple HTTP web server, delivering web pages on HTTP requests on a given port.

One of the reasons for using a third party application was to demonstrate that the implementation of our migration framework via AOP allows a non-intrusive extension of the original (binary) Web server application.

The aspect code for the migrating variant of the web server is a little more complex than the aspect code for the file version checker. The new web server uses a policy module that checks on request the CPU load on the local machine. One could also imagine that the policy module uses the IO load as trigger. The migration function call is performed through the aspect code after successfully answering a HTTP request.

The aspect code utilizes the *MigrateIfNeeded SearchHost()* call of our library. This variant locates the migration destination itself. The policy assembly must be available locally on every possible migration server instance in the network. With this preparation the mechanism of searching a destination through a broadcast policy check request can be used as expected.

Another part of the migrating web server is the PMH module, which implements forwarding of http requests. It is needed because the TCP listener socket is not transferable to the new node. Also the accessibility of the server at the new host can only be ensured with this architecture. The actual implementation of the PMH handler responds to incoming HTTP requests with a HTTP redirect message. This message contains the address of the new host. The HTTP client recognizes the 'forwarding information' message and sends the request again to new address.

The web server application is a good example for load balancing through migration. Figure 6 shows the client response times for a request in the migrated and the non-migrated case under varying load scenarios on the web server's host computer. We have used the tool *cpustres* from the Windows 2000 Resource Kit to simulate computational load on the Web server's host computer. This tool has been run with one to four active threads. All but one of *cpustres*' threads were spinning in a loop, using their scheduling quantum completely. Only one thread was varied in its CPU usage, using an adjustable amount of its quantum (10%, 40%, 60%, 90%, respectively). This is reflected in the horizontal scale of the diagram in Figure 6. Our experiment shows a response time of the migrating web server, which is initially by a factor of two better than the original web server's response time on a lightly loaded machine. However, with increasing load, there is a break even and finally, with very high load in the original web

[2] http://www.codeproject.com/csharp/mywebserver.asp

server's computer, the migrating version behaves worse than the original web server. This is due to the delays introduced by the post migration handler module (PMH proxy), which remains on the original node even if the web server migrates.

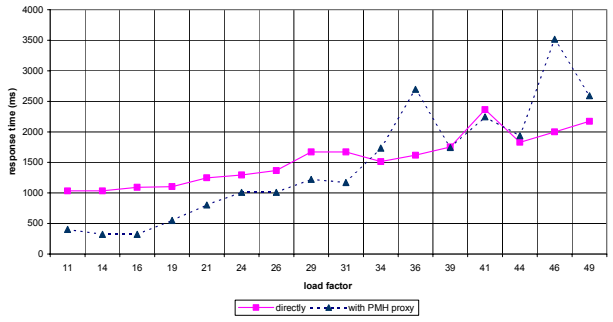

Figure 6: Web server response time under load

A central problem with the web server application was its multi-threaded architecture. The actual facility is not able not handle migrants with multiple threads. In the concrete case it was not possible to re-migrate the program from a host since the TCP listener thread could not be terminated explicitly in this situation. The only way to solve the problem was to unload the complete application domain of the migrant during the cleanup phase in the aspect code.

6. Conclusions

We have described the integration of a migration facility into the .NET framework. Using aspect-techniques for integrating migration into .NET addresses non-functional system properties on the middleware level, without the need to manipulate lower system layers like the operating system itself. We could demonstrate the extension of a binary .NET component (the Web server application) into a migrating version without being required to even see the Web server's source code.

There are several practical reasons to integrate migration in distributed environments, among them load balancing (explicit positioning of processes to distribute the computational load), and load sharing (automatic migration of computational intense tasks to idle machines).

The experimental migrating web server, one of our proof-of-concept applications, demonstrates the benefits of load balancing through migration. In comparison to the non-migrating version, it could shorten the response time to clients' requests by a factor of two on a lightly loaded machine. Evaluation of the migrating file version checker, our second proof-of-concept applications, led to the conclusion that inefficient cache usage and big state spaces of migrating applications have to be carefully considered as they may neglect performance benefits achieved by migration.

We are currently concentrating on a metrics, which takes those issues into account and will be implemented as part of the migration server's policy module.

References

[1] G. Attardi et.al. Techniques for dynamic software migration. In *Esprit '88, Proc. of the 5th Annual Esprit Conference*, pages 475-491. North-Holland, 1988.

[2] Paulo Amaral, Christian Jacquemot, Peter Jensen, Rodger Lea, and Adam Mirowski. Transparent object migration in COOL2. In Yolande Berbers and Peter Dickman, editors, *Position Papers of the ECOOP '92 Workshop W2*, pages 72-77, 1992.

[3] Yeshayahu Artsy and Raphael A. Finkel. Designing a process migration facility: The Charlotte experience. *IEEE Computer*, 22(9):47-56, 1989.

[4] N. Davies and G. Blair and J. Mariani. Supporting Persistent Re-locatable Objects in the ANSA Architecture. In *Internal Report Ref: MPG-92-04* and submitted for publication, Lancaster University, Bailrigg, Lancaster,# LA1 4YR, U.K., February 1992.

[5] IBM Corporation. Autonomic Computing manifesto, 2001. http://www.ibm.com/research/autonomic

[6] Adam John Ferrari. Process state capture and recovery in high-performance heterogeneous distributed computing systems. Dissertation, Faculty of the School of Engineering and Applied Science, University of Virginia, January 1998.

[7] Alfonso Fuggetta, Gian Pietro Picco and Giovanni Vigna, Understanding Code Mobility. In *IEEE Transactions on Software Engineering*, 24(5): 342-361,1998.

[8] David Halls. Applying Mobile Code to Distributed Systems. PhD thesis, University of Cambridge, 1997.

[9] Mark Nuttall. Survey of systems providing process or object migration. In *Imperial College Research Report* DoC 94/10, 1994.

[10] Mark Nuttall and Morris Sloman. Workload characteristics for process migration and load balancing. In *International Conference on Distributed Computing Systems*, 1997.

[11] Michael Philippsen and Matthias Zenger. JavaParty - transparent remote objects in Java. In *Concurrency: Practice and Experience*, 9(11):1225-1242, 1997.

[12] Wolfgang Schult and Andreas Polze. Aspect-Oriented Programming with C# and .NET. In *International Symposium on Object-oriented Real-time distributed Computing (ISORC)*, Crystal City, VA, USA. April 29 - May 1 2002, pages 241-248.

[13] Marc Shapiro, Philippe Gautron, and Laurence Mosseri. Persistence and migration for C++ objects. In Stephen Cook, editor, *ECOOP'89, Proc. of the Third European Conf. on Object-Oriented Programming*, pages 191-204, Nottingham (GB), 1989.

[14] Peter Smith and Norman C. Hutchinson. Heterogeneous process migration: The TUI system. In *Software Practice and Experience*, 28(6):611-639, 1998.