Communicating Process Architectures 2005
Jan Broenink, Herman Roebbers, Johan Sunter, Peter Welch, and David Wood (Eds.)
IOS Press, 2005

Towards Strong Mobility in the Shared Source CLI

Johnston STEWART [a], Paddy NIXON [b], Tim WALSH [c] and Ian FERGUSON [a]

[a] *SmartLab, Dept. of Comp. and Inf. Sciences, University of Strathclyde, Glasgow*
[b] *Computer Science Department, University College Dublin, Dublin 4, Ireland*
[c] *Department of Computer Science, University of Dublin, Trinity College, Dublin, Ireland*

{johnston.stewart, ian.ferguson}@cis.strath.ac.uk
paddy.nixon@ucd.ie, tim.walsh@cs.tcd.ie

Abstract. Migrating a thread while preserving its state is a useful mechanism to have in situations where load balancing within applications with intensive data processing is required. Strong mobility systems, however, are rarely developed or implemented as they introduce a number of major challenges into the implementation of the system. This is due to the fact that the underlying infrastructure that most computers operate on was never designed to accommodate such a system, and because of this it actually impedes the development of these systems to some degree. Using a system based around a virtual machine, such as Microsoft's Common Language Runtime (CLR), circumnavigates many of these problems by abstracting away system differences. In this paper we outline the architecture of the threading mechanism in the shared source version of the CLR known as the Shared Source Common Language Infrastructure (SSCLI). We also outline how we are porting strong mobility into the SSCLI, taking advantage of its virtual machine.

Keywords. Thread Mobility, Virtual Machine, Common Language Runtime.

Introduction

The term *mobility*, when applied to code, has two guises: *weak mobility* and *strong mobility*. Weak mobility [1] is defined as the ability to allow code transfer across nodes – the code can be accompanied by some initialization data, but the execution state cannot be migrated. Strong mobility [1] is defined as the ability to migrate both the code and the execution state to a remote host. Weak mobility is favoured for most mobile systems in the commercial domain as it is relatively easy to design and implement such a system. Strong mobility systems on the other hand are rarely developed or implemented as they introduce a number of major challenges for the implementation of the system. This is due to the fact that the underlying infrastructure that most computers operate on was never designed to accommodate such a system, and because of this it actually impedes the development of these systems somewhat. Using a system based around a virtual machine, however, circumnavigates the problems introduced by varying underlying operating systems and system architectures. We propose to augment the SSCLI's thread model to allow the stack of a thread to be captured, saved and then reseeded and resumed on a new node. Doing this in a safe and managed manner is a challenging proposition and is necessary for the implementation of closures and continuations.

The SSCLI is a portable implementation of the programming tools and libraries that comprise the ECMA-335 Common Language Infrastructure standard [2]. The ECMA (European Computer Manufacturers Association) CLI is a standardised specification for a virtual execution environment with virtual execution occurring within the CLI under the control of its execution engine (EE).

The ECMA CLI describes a data-driven architecture, in which various components in various languages can be brought together to form self-assembling, typesafe software systems. This process is driven by metadata, which is used by the developer to describe the behaviour of the software and is used by the execution engine (EE) to allow the safe loading of managed components from different sources. The EE (often referred to as the runtime) hosts components by interpreting the metadata which describes them at runtime.

On the one hand the CLI execution engine is similar to an operating system, it is a privileged piece of code which provides services and managed resources for code executing under its control. Programs can explicitly request services or they can be made available as a part of the execution model. At the same time the CLI is similar to the traditional model of compiler, linker and loader performing in-memory layout, compilation and symbol resolution.

The CLI enables seamless sharing of computing resources and responsibilities, allowing unmanaged code to coexist safely with managed code. The SSCLI (also known as Rotor) consists of a fully functional CLI execution engine, a C# compiler, essential programming libraries and a number of development tools, built using a combination of C++ and C# and a small amount of assembler for processor specific details [2].

1. Motivation

It is the belief of the authors that a coherent implementation of thread migration will be a foundational step for more advanced agent related research and is a core requirement for the implementation of ambient style systems.

Many applications could benefit from the possibility of migrating threads between machines. Safety critical applications for instance often cannot afford to be shut down for any reason. If running threads could migrate to another host temporarily then maintenance and upgrades could occur without disruption to the service.

Some of the main advantages of mobile computation [3] are noted below:

- *Load sharing:* Distributing computations among processors around the system can lighten the load on heavily used processors and make use of underused resources.
- *Communications performance:* Active objects which interact intensively can be migrated to the same node to reduce the communication overheads for the duration of their interaction.
- *Availability:* Objects can be moved to different nodes in order to improve service and provide better protection against failure and lost connections.
- *Reconfiguration:* Migrating objects allow continued service during scheduled downtime or node failure.
- *Resource utilisation:* An object can visit a node in order to take advantage of services or capabilities at that particular location.

2. Threads in the SSCLI

Thread structures within the SSCLI provide a method for associating the microprocessor's execution stack with related runtime data (by adding a chain of execution engine frames to the stack before JIT code is executed). This data includes security information, garbage collection markers and program variables as well as other information [2]. The logical abstraction of a thread of control is captured by an instance of the `System.Threading.Thread` object in the class library, [4] and is known as a managed thread.

2.1 PAL Threads

Within the execution engine (EE) managed threads are implemented on top of Platform Adaptation Layer (PAL) threads. This is done in order to abstract away the threading details, and differing semantics of threading, from differing implementations of the threading model used by different operating systems that the SSCLI can run on [5] [2]. The PAL hides these differences beneath a single set of APIs. PAL threads themselves have a one to one relationship with OS threads (this does not, however, imply a one to one relationship between managed and unmanaged threads) [2].

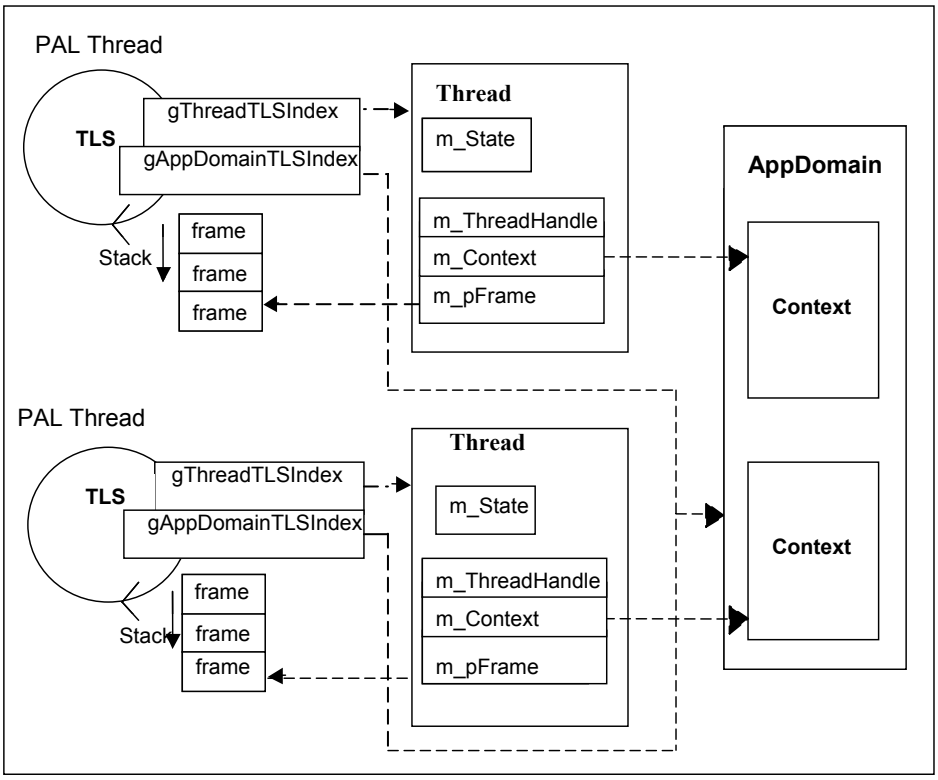

Figure 1. Overview of the threading mechanism

Managed threads are wrapped around PAL threads, and always have an associated PAL thread, however, a PAL thread need not always have an associated managed thread. Threads which originate from outside of the Common Language Runtime (CLR) are known as unmanaged threads [2]. PAL threads need to be able maintain private, per-thread state so that the EE can track specific threads by associating a Thread instance with the underlying PAL thread. This is achieved by using the `m_ThreadHandle` field (a private attribute) in the Thread type, which contains a `HANDLE` to the PAL thread (see Figure 1), in order to control and schedule execution on the thread.

There is no PAL call to enable navigating from a thread handle to a managed thread, so the EE maintains a `ThreadStore` which can be used to enumerate managed threads from either managed or unmanaged code (Figure 1) [2, 6].

2.2 Interoperation of Managed and Unmanaged Code

The CLR allows managed code to call unmanaged code and vice versa and allows managed and unmanaged code to interoperate freely and managed threads mix the execution state of managed and unmanaged code on a single stack (Figure 2) [2]. Also, a managed process can contain many different threads of control (Figure 2). The EE uses PAL threads which become associated with managed code, to maintain exception handlers, scheduling priorities, and a set of structures that the underlying platform uses to save the context (which contains the values held in the machine registers and the state of the current execution stack) whenever it pre-empts the thread's execution [2]. Basically the thread context contains all of the information that the thread needs in order to seamlessly resume execution [7].

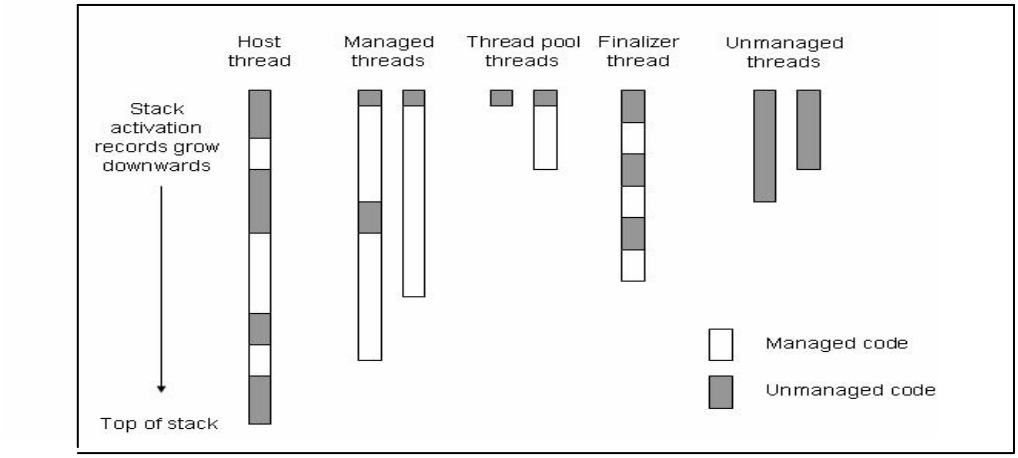

Figure 2. Threads within a managed process

Transitions between managed and unmanaged code can be created in many ways. Within managed code, application boundaries or remoting contexts can be crossed, security permissions can change and exceptions can be thrown and in all of these cases isolation needs to be maintained. [2] When a PAL thread attempts to enter managed code an instance of the Thread class is set up to wrap it using the `SetupThread` method. Two checks are made, a call to `GetThread` is made to look for a cached Thread instance in the Thread Local Storage (TLS), a feature which allows a PAL consumer to associate data with a specific thread [2], in order that the calling PAL thread can make sure the PAL thread is not already known to the EE. Next `SetupThread` ensures that the call is not coming from a different thread than the thread being initialised, by checking the `ThreadStore` for a matching identifier. If an identifier is found `SetupThread` will return, if not the PAL thread is unknown to the EE and a new Thread object is created, installed on the PAL thread's TLS, and marked as started [2]. The EE then adds the new Thread instance to the `ThreadStore's` list of all the threads ever seen. Threads that wander into the EE which are not previously known are set as background threads [2]. The easy way to control the creation and scheduling of threads is to let the CLR do it, using the thread pool.

2.3 The Thread Pool and Separate Threads in the SSCLI.

The SSCLI includes a pooling mechanism which caters simply for multithreaded scenarios. The `ThreadPool` class provides a fixed pool of worker threads to service incoming requests [2]. This class is provided within the SSCLI so that systems like Windows 98, which does not have native support for thread pools, can still utilise them [8]. Initially all of the thread pool threads are idle, then when a request comes in the system looks up the thread pool, finds an idle thread and assigns that thread to the request which is to be serviced. If all of the threads in the thread pool are busy when a request comes in the system either grows the thread pool by one and assigns the new thread to the request or, if the thread pool is already operating with its maximum amount of worker threads, waits for a thread within the pool to become free [9]. Each thread in the thread pool runs at the default priority and uses the default stack size [10].

There is a single managed thread pool per process, access to which is gained through the `ThreadPool` class [2]. Using the thread pool is the easiest way to code for tasks which are required to handle multiple threads for relatively short tasks without blocking other threads and where there is no need for a specific order of scheduling for the tasks performed. However, the thread pool is not a good choice if the task to be performed needs to be set at a specific priority or if it might run for an extended period of time, as this would block other threads [7]. The thread pool is also of no use if a stable identity has to be associated with the thread (so that it can be suspended or discovered by name). Even when using the c++ method `GetThreadInfo` there is no guarantee that the `threadID` returned is the correct Win32 `threadID` of the underlying OS thread, as the thread pool is abstracted one layer away from the Win32 `threadID` [11]. In order to retain some degree of control over a managed thread's scheduling, and to be able to accurately discover its associated OS thread, separate instances of the Thread class must be used.

The alternative to using the threads from the thread pool is to create separate instances of the Thread class. Management of all threads within the CLR is done using the thread class, this includes threads created by the CLR, and any threads which originate outside of the runtime that enter the managed environment in order to execute code [2].

2.4 Managed Thread to OS Thread Relationship

In the current implementation of the CLR, an OS thread will have only one managed thread associated with it in a given Application Domain, known as an `AppDomain`, which is a form of isolation which the runtime uses to ensure that code running in one application cannot affect other applications. AppDomains provide a secure and versatile unit of processing that the CLR uses to isolate applications from each other [12]. If an OS thread executes code in more than one `AppDomain`, each `AppDomain` will have a distinct managed thread associated to that thread (see Figure 3) [13].

The threads within the CLR may or may not have a corresponding OS thread, as threads which have stopped or which have been created but not started, do not have a corresponding OS thread. Also, if an OS thread has not yet executed any managed code there will not be a managed thread object corresponding to it [2].

Within managed threading, `Thread.GetHashCode` is the stable identification for a managed thread and this will not collide with the value of any other thread for the lifetime of the thread which it identifies, regardless of the application domain from which the value is obtained.

An OS `ThreadId` has no fixed relationship to a managed thread, as many managed threads may be serviced by the same OS thread (Figure 4) [5].

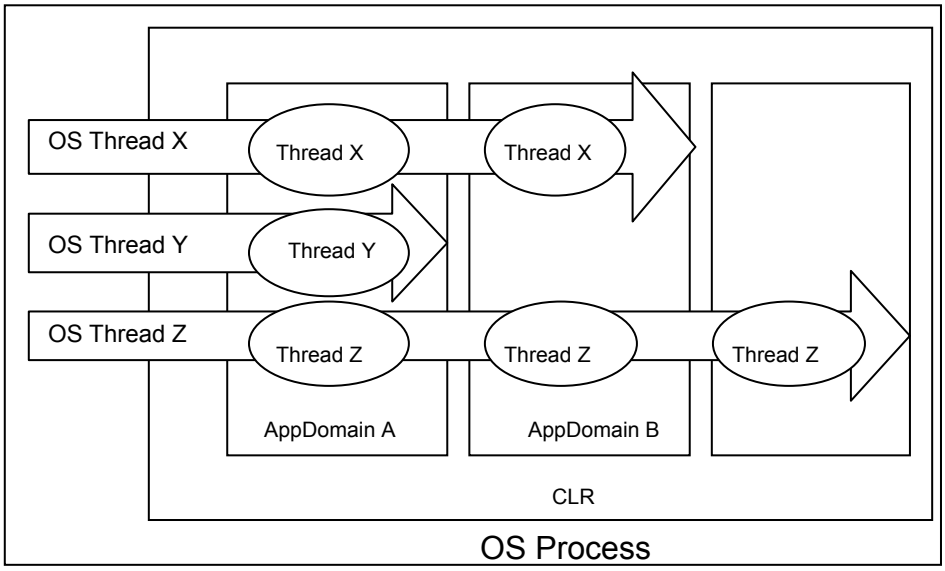

Figure 3. AppDomains and Threads

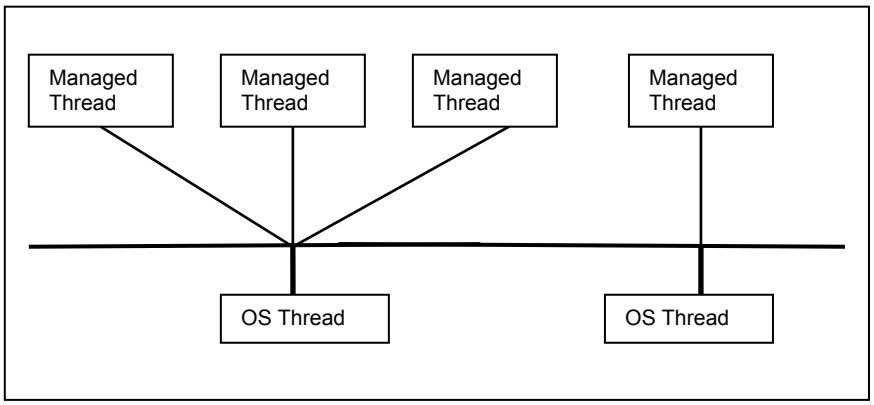

Figure 4. Managed thread to OS thread relationship

2.5 Overview of the Thread Class

The `Thread` class creates and controls a thread, sets its priority, and gets its status. The `Thread` type is safe for multithreaded operations. The `ThreadPriority` property can be used to schedule a priority level for a thread, however, this is not guaranteed to be honoured by the underlying operating system [5]. A managed thread can be `start`ed, `join`ed, `suspend`ed and `kill`ed. It is exposed and used as a managed component although much of the implementation behind it is written in C++ and is internal to the EE. The code used to represent managed instances can be found in the C++ class `Comsynchronisable.cpp` [2].

2.6 Functionality of the Thread Class

The thread class does not have a method with which the thread ID of an OS thread can be discovered, as the ECMA standard 335 states that the Thread class represents a logical thread and not necessarily an operating system thread as explained above. The OS thread ID can be discovered, along with other information about the physical thread, by using the `System.Diagnostic.ProcessThread` class, or `AppDomain.GetCurrentThreadId`

method which is a wrapper built around the Win32 `GetCurrentThreadId` function [14]. The managed threads within the CLR map loosely to the win32 threads in the underlying OS (Table 1) [15]; however, this does not represent identical functionality as some methods or functions have no equivalent. The Thread class itself provides the other methods needed to create and control a thread object.

Table 1. Functionality mapping between Win32 and CLR

Win32	CLR
`CreateThread`	Mix of `Thread` and `ThreadStart`
`TerminateThread`	`Thread.Abort`
`SuspendThread`	`Thread.Suspend`
`ResumeThread`	`Thread.Resume`
`Sleep`	`Thread.Sleep`
`WaitForSingleObject` on thread handle	`Thread.Join`
`ExitThread`	No equivalent
`GetCurrentThread`	`Thread.CurrentThread`
`SetThreadPriority`	`Thread.Priority`
No equivalent	`Thread.Name`
No equivalent	`Thread.IsBackground`
Close to `CoInitializeEx` (OLE32.DLL)	`Thread.ApartmentState`

3. Thread Local Storage

With respect to managed threads, thread local storage (TLS) provides dynamic data slots unique to a thread and `AppDomain` combination which are used to store thread specific data. There are two types of data slots; named and unnamed. Named slots can be given a mnemonic identifier, however, other components can then (intentionally or not) modify them by using the same name for their own thread relative storage. If an unnamed slot is not exposed to any other code it cannot be used by any other component [16].

PAL threads have their own TLS which allows a PAL consumer to associate data with a specific thread for later retrieval in the thread's context. In unmanaged code the TLS slot is utilised by calling `TLSAlloc`. This slot can then have its value set by `TlsSetValue` (which stores a pointer to the allocated memory) or retrieved using `TlsGetValue` (which returns the pointer to the thread's memory slot). This memory is freed on thread termination [2].

The CLR maintains a lot of information within the PAL thread's TLS, in particular references to the current `AppDomain` and managed `Thread` object. Whenever a PAL thread crosses from one `AppDomain` to another, the CLR adjusts the references in the TLS to point to the new current managed thread and `AppDomain`. The current implementation of the CLR maintains a per-`AppDomain` table which ensures that any PAL thread is associated with only one managed thread object per `AppDomain` [13].

4. Method State

The CLI manages multiple concurrent threads of control (which are not necessarily the same as the threads provided by the host operating system), multiple managed heaps and a shared memory address space (Figure 5) [4]. A thread of control can be thought of as a singly linked list of method states (see Figure 5), where a new state is created and linked back to the current state by a method call instruction, then removed when the method call completes (by a normal return, a *tail-call*, or an exception).

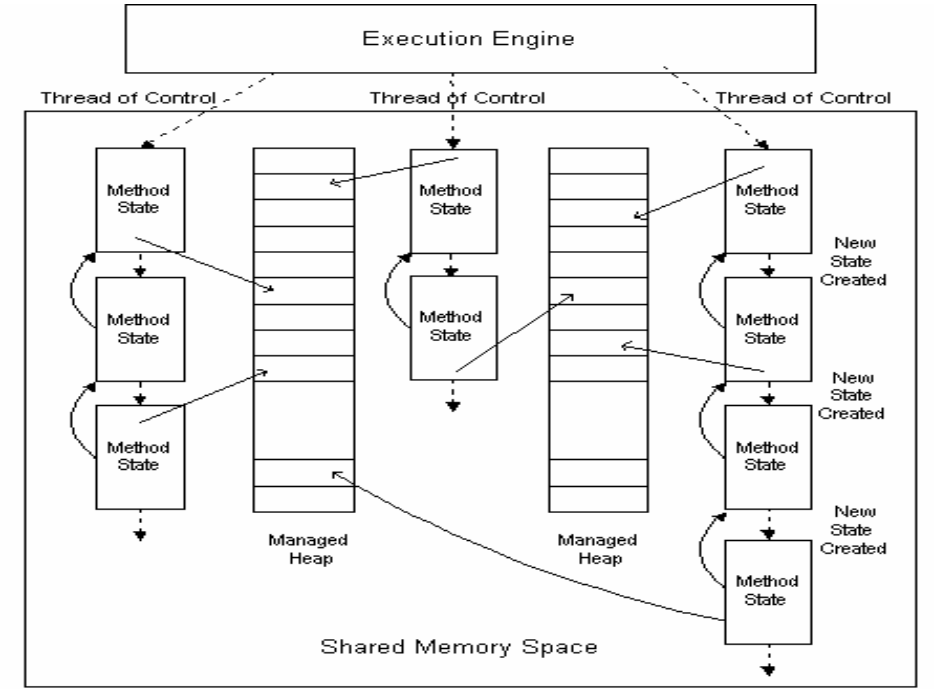

Figure 5. The machine state model, including threads of control, method states, and multiple heaps in a shared address space.

Method state describes the environment within which a method executes. In conventional compiler terminology, the method state corresponds to a superset of the information captured in the invocation stack frame.

The CLI method state [4] (Figure 6) consists of the following items; an *instruction pointer* (**IP**), an *evaluation stack*, a *local variable array* (starting at index 0), an *argument array*, a *methodInfo* handle, a *local memory pool*, a *return state* handle and a *security descriptor*. The method state's four areas; incoming arguments array, local variables array, local memory pool and evaluation stack, (Figure 6) are specified as if logically distinct areas [4].

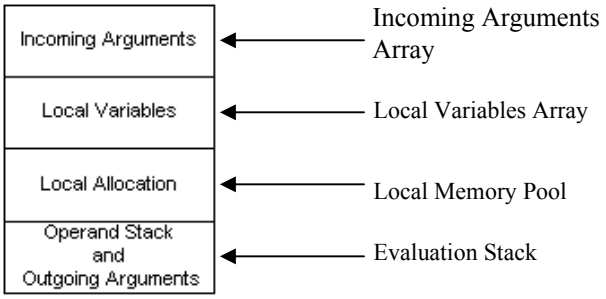

Figure 6. Method State

5. SSCLI Stack Management for Managed Threads

There is only one physical stack per process and all the threads within a process share this stack, using their linked list of stack frames (as explained above) to simulate a logically separate stack. This means that although the threads serviced by a single PAL thread all have their frames on the same stack, they can be thought of as having their own 'virtual' stack. These stack frames will be referred to as activation records in accordance with the Microsoft convention. Once a PAL thread has a `Thread` instance associated with it, managed code can be executed on it. Before any JIT compiled code is executed two important tracking structures are pushed onto the stack; an exception handler, to wrap the unmanaged code, and a chain of execution engine frames, to annotate parts of the stack with runtime information produced by the EE. A service called the code manager is used to deal with the intricacies of tracing the stack, and joins the managed and unmanaged portions of the stack into a single coherent view. When it is needed the information is gleaned from the stack by traversing the stack's call chain to extract the current execution state in what is known as a stackwalk. A linked list of EE frames (instances of the `Frame` class or one of its subclasses) is associated with each managed `Thread` object [2].

6. Towards Strong Mobility in the SSCLI

The advantage of strong mobility is that long-running or long-lived threads can suddenly move or be moved from one host to another. A transparent mechanism such as this would allow continued thread execution without data loss in its ongoing execution. This approach is useful in building distributed systems with complex load balancing requirements where the threads involved need not know about their movement. Strong mobility has disadvantages also, i.e. if a particular resource on the original node is necessary for the threads execution this will cause problems at the destination node. Using containers [17] allows us to package necessary objects with the executing thread and send it during the migration.

6.1 Containers

The container abstraction closely associates threads and their corresponding data objects (figure 7) and are the unit of migration. Containers may also contain passive objects for retaining data and results. Containers execute in 'homes', which act as a sandbox and receive containers and allow them to resume execution. Using a container allows us to

group threads and their respective objects into a single structured unit for migration [17]. Using appropriately set up `AppDomains` as containers and the CLR's virtual machine as a 'home' gives us a starting point for developing strong mobility in the SSCLI. Due to the fact that the SSCLI was written with distributed applications in mind, crossing `AppDomain` boundaries is a relatively simple task. This action of crossing the boundaries of the `AppDomain` would quickly break an application which seeks to contain the inhabitants of a specific `AppDomain` and disallow entry to entities outside of the `AppDomain`. In view of the fact that code can be written in this way, we currently rely on the code being written in a responsible way in order to avoid breaking the application.

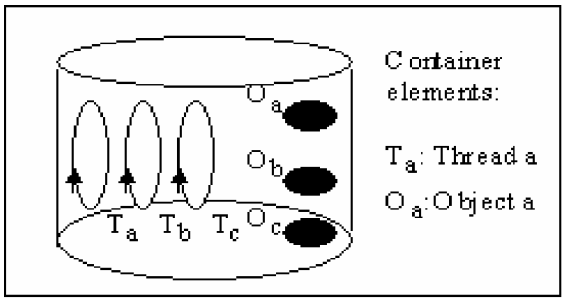

Figure 7. Containers

6.2 Thread Migration

Our solution is currently a work in progress based around using the SSCLI's AppDomain as a container and a modified version of the simple migration system known as Migrants [18] to facilitate transfer of the container from one node to another. Managed threads in the CLR are objects, however. They wrap PAL threads which in turn sit on top of OS threads, and because of this they are deliberately not *serializable* [19]. This means that it is not possible to simply serialize the thread in order to migrate the container to the remote host. At present the threading mechanisms within the SSCLI provide the basic functionality with no mechanism for state capture or thread migration [20]. We can, however, use the Win32 API in order to discover the managed thread's underlying OS thread and then use this information to save that particular thread's data from the OS thread's stack. The threads can then be serialized and sent to the remote host with their instance variables and code where they will be unpackaged and the container can then be reseeded and execution resumed.

7. Conclusions

Using the abstraction of a container within the SSCLI, utilising its virtual machine and taking advantage of `AppDomains`, makes thread migration with strong mobility a possibility. The initial implementation will be easily broken; however, this is something which can be made more robust over time. It can be argued that the proposed system is not one of true strong mobility, as references outside of the container cannot be supported. However, this can be seen as the first step to instantiating usable, useful strong mobility (for such areas as load balancing).

Acknowledgements

This research was supported by Microsoft Research and the University of Strathclyde.

References

[1] A. Fuggetta, G.P. Picco, G. Vigna, *Understanding Code Mobility*, IEEE Trans of Software Eng., vol 24, no. 5, pp. 342-361, May 1998.
[2] David Stutz, Ted Neward and Geoff Shilling. *Shared Source CLI Essentials.* O'Reilly 2003 First Edition.
[3] E. Jul, H. Levy, N. Hutchinson, A. Black, Fine-Grained Mobility in the Emerald System, ACM Trans. On Computer Systems, Vol. 6, no. 1, pp. 109-133, February 1988.
[4] Common Language Infrastructure (CLI) Partition I: Concepts and Architecture. *ECMA TC39/TG3.* Final draft – October 2002.
[5] Microsoft Corporation, *Thread Class Overview*, `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystemthreadingthreadclasstopic.asp` – accessed 02/08/05.
[6] Jason Whittington, *Inside Rotor Presentation*, `http://staff.develop.com/jasonw/tools_rotor_2002.ppt` – accessed 02/08/05.
[7] Microsoft Corporation, *Threads and Threading*, `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconthreadsthreading.asp` – accessed 02/08/05.
[8] Jeffrey Richter, *The CLR's Threadpool*, `http://msdn.microsoft.com/msdnmag/issues/03/06/NET/` accessed 02/08/05.
[9] Microsoft Corporation, *Threadpooling*, `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconthreadpooling.asp` – accessed 02/08/05.
[10] Microsoft Corporation, *Threadpool Class*, `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfSystemThreadingThreadPoolClassTopic.asp` – accessed 02/08/05.
[11] Brian Dowds, *Introduction to the CLR*, `http://support.microsoft.com/default.aspx?scid=%2Fservicedesks2Fwebcasts%2Fwc022802%2FWCT022802.asp` – accessed 02/08/05.
[12] Microsoft Corporation, *AppDomain Class*, `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystemappdomainclasstopic.asp` – accessed 02/08/05.
[13] Don Box with Chris Sells. *Essential . NET volume 1*. Addison Wesley 2003
[14] Dino Esposito, Windows *Hooks in the .NET Framework*, `http://msdn.microsoft.com/msdnmag/issues/02/10/cuttingedge/` – accessed 02/08/05.
[15] Microsoft Corporation, *Managed and Unmanaged Threading in Microsoft Windows*, `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconmanagedunmanagedthreadinginmicrosoftwindows.asp` – accessed 02/08/05.
[16] Microsoft Corporation, *Thread Local Storage and Thread-Relative Local Fields*, `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconthread-localstoragethread-relativestaticfields.asp` – accessed 02/08/05
[17] Tim Walsh, Paddy Nixon, Simon Dobson, *As strong as possible mobility: An Architecture for stateful object migration on the Internet*. Technical Report TCD-CS-2000-11, Department of Computer Science, Trinity College Dublin, 2000.
[18] Simon Dobson, Migrants – *A Brain-Numbingly Simple Mobile Code System*, 2003, `https://www.cs.tcd.ie/Simon.Dobson/software/migrants/index.html`
[19] SSCLI source code. Thread class. `sscli\clr\src\bcl\system\threading\thread.cs`
[20] Platt, David, *Introducing Microsoft .NET*, Microsoft Press, 2001.