Amadeus Project

Micro-kernel Support for Migration

Martin O'Connor, Brendan Tangney, Vinny Cahill and Neville Harris,
Distributed Systems Group,
Dept. of Computer Science, Trinity College, Dublin, Ireland

Distributed Systems Group
Department of Computer Science
University of Dublin
Trinity College, Dublin 2, Ireland.
Fax: +353-1-6772204

Document Status Public
Distribution DSG Internal
Document # TCD-CS-94-43
Publication To appear in Distributed Systems Engineering Journal

Micro-kernel Support for Migration

Martin O'Connor[*], Brendan Tangney, Vinny Cahill and Neville Harris[†]
Distributed Systems Group,
Dept. of Computer Science, Trinity College, Dublin, Ireland

Abstract

This paper aims to show that micro-kernels are a more suitable environment for the implementation of process migration than conventional monolithic kernels. This premise is illustrated with an implementation of migration on top of the CHORUS micro-kernel. The implementation emphasises the modularisation of the various components of a migration implementation, as well as drawing a strong distinction between policy issues and migration mechanisms. This modularisation serves to illustrate the functionality required of a micro-kernel to support migration. We draw two additional conclusions from this implementation. Firstly, we conclude that migration should not be provided by the micro-kernel itself, but should instead be implemented at a level just above it. And, secondly, we identify a very small number of areas where changes may be required to existing micro-kernel interfaces to facilitate the implementation of migration at this level.

1 Introduction

A number of distributed operating systems have demonstrated the viability of process migration. The most significant implementations include the V-System [21], Charlotte [3], Sprite [8], DEMOS/MP [18], LOCUS [22], Accent [24] and, more recently, Mach [16]; over twenty other systems have been described in the literature [10]. However, many of these systems have made considerable simplifying assumptions about their environment in order to implement migration and few could be said to provide truly transparent migration. For example, many systems do not support migration of processes that have knowledge about their environment, such as the name of the host on which they are running. Other restrictions include the lack of support for processes that use interprocess communication facilities and the inability of migrant processes to spawn subprocesses. Restrictions may also be placed on processes using special files, non-standard buffers, signal catching, and file locking.

While many of these systems were used extensively for research purposes, only one system known to the authors has evolved into a commercial implementation [22]. This dearth of commercial migration systems and the restricted utility of many existing systems can at least in part be attributed to the considerable implementation difficulties that accompany migration. Many of these difficulties arise because migration is typically added to an operating system as an afterthought. It is rarely included as an integral system component from the operating system's inception. The heavy interactions typical between any migration system and the underlying operating system often ensure that this later addition of migration is problematic.

Transparency and encapsulation have been identified as the key indexes to the amount of effort required to implement migration in a system. Both of these issues have a considerable influence on the success of a migration implementation. We discuss both in some detail before considering them in the context of micro-kernels.

1.1 Transparency

Transparency generally refers to the concealment of particular details from the user or application programmer. A distributed system may, for example, try to conceal the fact that it is distributed and aim to provide the same interface as a single-node system. In the context of migration, transparency generally refers to the ability to move processes without the process itself nec-

*Author's current address: IBM T.J. Watson Research Center, Yorktown Heights, NY
†Email: Brendan.Tangney@dsg.cs.tcd.ie

essarily being aware of this movement. In general, the greater the degree of distribution transparency provided by an underlying system, the easier one can add migration to that system. As a consequence, successful migration facilities are encountered mostly in transparently distributed environments [18, 21]. A high price may have to be paid when providing migration in a system that does not exhibit a large degree of transparency. This price normally makes itself felt in the creation of *residual dependencies*.

A residual dependency is the dependence by a migrated process on its initial node or on any of the nodes it visited previously. Residual dependencies are caused by incomplete transfers of process state. Typically, such dependencies arise because some process state information would not remain meaningful across a migration and hence cannot be moved with the process. This state information is tied in some way to the source node of the migration. Residual dependencies often arise because the underlying system is not fully suitable for the implementation of a migration facility. For example, *links* and *ports* were used in DEMOS/MP [18] and Accent [24] to provide a level of indirection when sending a message to a process; the Sprite operating system [8] does not provide this level of indirection - it uses an RPC-based communication system - so a transparency-violating *home node* is necessary to deal with location-dependent calls.

Clearly, the greater the degree of transparency provided by the system the fewer such dependencies will arise. However, no systems offer complete freedom from residual dependencies regardless of the amount of transparency provided. Even if a process has a transparent reference to all servers that it interacts with, those servers are typically not migrated with the process, so residual dependencies still arise.

The requirement of reducing residual dependencies emphasises the need to minimize the amount of state outside a process on which it is dependent. The reduction of residual dependencies is one of the trickiest issues in process migration.

1.2 Encapsulation

Before a process can be migrated its state must be encapsulated. This encapsulation must extract all the state information from a process and assemble it in a structure that can be shipped between machines. This state information includes all internal process state variables and may include details of the process's interactions with other system components. Collecting this state information in one place may prove difficult. The most common source of difficulty arises because this state may be distributed throughout numerous operating system data structures. Obtaining a consistent snapshot of this state adds an extra layer of difficulty.

Clearly, a high degree of process encapsulation is required if migration is to be provided efficiently. For example, encapsulation is difficult in classical implementations of UNIX[1] [1, 5, 13, 14] because these systems hold process state throughout various kernel tables. In addition, keeping this state consistent while it is being gathered is non-trivial.

1.3 Micro-kernels

Modern micro-kernels often offer very high degrees of both encapsulation and transparency [9, 20]. Many kernels, for example, enforce a level of indirection between physical and logical IPC addresses, the most typical mechanism being ports. Thus, if a process sends its messages to location-independent logical addresses, rather than to physical addresses, the operating system may transparently rebind these references when a process migrates, thus freeing the process from ties to a particular location. In addition, many implementations aim to ensure that any interactions between a process and its environment are performed via message-exchange. Ideally, they guarantee that all interaction outside a process's address space should be performed using network-transparent inter-process communication primitives.

With these increased degrees of transparency and encapsulation, migration should be significantly less difficult to implement on micro-kernels. This paper aims to prove that this is the case. This premise is illustrated with an implementation of migration on top of the CHORUS[2] micro-kernel. We draw two additional conclusions from this implementation. Firstly, we conclude that migration should not be provided by the micro-kernel itself, but should instead be implemented

[1] UNIX is a trademark of UNIX Systems Laboratories, Inc.

[2] CHORUS and CHORUS/MiX are trademarks of Chorus Systemès.

at a level just above it. And, secondly, we identify a very small number of areas where changes may be required to existing micro-kernel interfaces to facilitate the implementation of migration at this level.

It should be noted that a monolithic kernel exhibiting high levels of transparency and encapsulation is also an eminently suitable candidate for migration. However, if we accept that the defining characteristic of a micro-kernel is that is provides a small set of abstractions from which all services are composed then, by definition, encapsulation is improved. A similar argument can be applied to transparency since there is only a small well-defined set of communication primitives which are designed to be extensible across a network. In addition the design of the virtual memory management subsystems typically provided by micro-kernels greatly enhances their suitability for migration. Many systems, for example, provide network-transparent page access, greatly simplifying the transfer of process address spaces in migration. Hence, it can be argued that micro-kernels by their nature are more suited to process migration.

1.4 Roadmap

The layout of this paper is as follows. The following section describes the design of a process migration system for a micro-kernel that aims to exploit the degrees of transparency and encapsulation provided by such systems. Section 3 describes an implementation of this design on the Chorus micro-kernel. A discussion of the lessons learnt from this implementation is given in section 4. We close with a summary of our conclusions.

2 Architecture of a Migration System

The issues presented by process migration have been addressed by a diversity of approaches, the approach adopted being strongly influenced by the underlying operating system. As a consequence of this diversity, simple abstractions on which a general migration design may be based do not easily present themselves. The aim of this section is to attempt to identify these primary abstractions and to describe a process migration architecture that provides for their clean separation in an implementation on a micro-kernel. The design attempts to avoid specifying particular implementation details. Instead, it defines a structure within which the implementation may be carried out. A similar attempt to separate implementation details from policy issues was made by the designers of the Charlotte migration system [2].

Operation	Description
`GetConfiguration`	Get kernel configuration
`GetStatistics`	Get load information
`Encapsulate`	Encapsulate a process
`Reestablish`	Ressurrect a process

Table 1: Kernel Module Interface

The design focuses on three areas: process encapsulation and re-establishment, migration policies, and state shipment. These issues are separately dealt with by three modules in the design. The modules are: the *kernel module*, the *policy module*, and the *transport module*. Each module is self-contained and inter-module interaction is governed by well-defined interfaces. Figure 1 illustrates a typical distribution of modules in a system.

The decision to migrate a process is initiated by the policy module based on the information it has obtained from kernel modules on participating machines. If, based on this information, a policy module decides to migrate a process, it will request the transport module to ship the process state. The following subsections detail the operation of these three modules.

2.1 Kernel Module

The *kernel module* abstracts the low-level issues involved in the encapsulation and re-establishment of processes. It is responsible for encapsulating the state information of migrant processes and for re-establishing these processes on other nodes using this state. This module is also responsible for providing load information to policy modules to enable them to make migration decisions.

Table 1 details a sample interface to a kernel module. This interface could be provided by the addition of extra system calls. The `GetConfiguration` call is provided to enable policy modules to determine the configuration of the ker-