Towards Safe and Efficient Customization in Distributed Systems

Position Paper

C. Bryce V. Issarny G. Muller I. Puaut

INRIA / IRISA
Campus de Beaulieu
35042 Rennes Cédex, France
Email:{bryce,issarny,muller,puaut}@irisa.fr

1 Introduction

Progress in microprocessor and communication system technology is leading to the emergence of new large scale distributed architectures whose power increases proportionally to the number of processor elements. Such architectures support execution of traditional distributed applications based on the client-server paradigm as well as parallel applications. Among other consequences, this framework requires the design of distributed operating systems that exhibit good performance for a wide range of applications. One way to achieve this consists of weakening the semantics of system functions, and to customize them to implement stronger semantics that match the applications' needs. Customization in distributed systems must be achieved in a safe and efficient way. The safety property relates to the fact that a customized function must implement a stronger semantics than that of the base function. Efficiency of a customization mechanism is evaluated in terms of the performance penalty resulting from the invocation of a customized function instead of the base function; ideally, the penalty should equal the execution cost due to the added functionality as if it were directly supported by the underlying distributed system.

This paper addresses support for safe and efficient customization in distributed systems at the design and implementation levels. Linguistic paradigms aimed at facilitating the design of safe customization are sketched in Section 2. In particular, we argue that behavioral subtyping, used in strongly-typed, object-oriented programming languages, helps the user to achieve safe customization: it can be used to reason about the strengthening of function semantics. Section 3 then addresses operating system support for implementing safe and efficient customization. Finally, Section 4 gives an overview of our current and future work that relates to the design and implementation of both linguistic and operating system paradigms aimed at safe and efficient customization within distributed systems.

2 Paradigms for the Design of Safe Customization

The object-oriented paradigm appears as a convenient programming support for customization (e.g., see [9]) : object definitions allow the system to be decomposed into a set of abstractions that can be customized, and the inheritance facility provides a tool for customizing existing abstractions. However, to ensure the correctness of customization with respect to the abstractions' semantics, the base object-oriented programming model must define a relation through the introduction of fault domains ensuring that the semantics of a customization $\mathcal{C}$ of an existing system function $\mathcal{S}$ is stronger than the one of $\mathcal{S}$. Subtyping that is especially present in strongly-typed, object-based programming languages provides the adequate basis for that framework.

Subtyping is a relation of inclusion between types that allows the use of an object of type T in place of an object of type U if T is a subtype of U. Existing definitions of subtyping relations can be divided into two categories depending on whether they focus on the syntactic or semantics notion of subtyping. The syntactic approach (e.g., see [7]) leads to define a subtyping relation that places constraints on the signatures of an object's methods; and hence allows to check at compile-time for any method call that the method is provided by the called object and that the actual parameters are assignable to the formal ones. The semantics approach leads to a definition of the subtyping relation that places additional constraints on an object's methods; these relate to the method's behavior as described in the corresponding type's specification (e.g., see [11]). The resulting subtyping is qualified as *behavioral* and allows to check whether properties about supertype values and methods are preserved when designing a subtype.

Behavioral subtyping provides a convenient framework to verify the correctness of customization from the standpoint of the objects' semantics. Given a system object $\mathcal{S}$, a customization $\mathcal{C}$ of $\mathcal{S}$ is correct if $\mathcal{C}$'s type is a subtype of $\mathcal{S}$'s type according to the underlying behavioral subtyping relation. More largely, a strongly-typed, object-oriented programming language integrating a behavioral subtyping relation constitutes an adequate basis for the development of trusted customization code. The next section addresses safe and efficient implementation of customization assuming trusted customization code; more precisely, it examines the needed operating system support to meet this goal.

3 Paradigms for Safe and Efficient Implementation of Customization

The code implementing a customized system function may be executed in one of the following system entities:

(1) the kernel;

(2) a separate server; assuming a distributed system based on the micro-kernel technology, the server may be either:

  (2.1) a supervisor actor as in Chorus [14], or

  (2.2) a server task running in user mode, as in Mach [1];

(3) a user's task where the code will execute either:

  (3.1) in the same task (and thus protection domain), or

(3.2) in a separate protection domain within the task's address space[1].

We assume that the code of the customized function is trusted. In the remainder of this section, we examine the degree of safety and efficiency offered by each of the above solutions for implementing customized functions.

Solution (1) consists of executing customization code within the kernel. The code is generated by dynamic code synthesis and is then linked into the kernel [12]. This approach has been used to perform efficient argument-marshaling in an RPC system [15]. The major advantage of code synthesis is its efficiency; the synthesized code is directly accessible through a kernel trap. On the other hand, its major drawback is that it is not scalable: the size of the kernel's address space bounds the number of procedures that can be integrated within the kernel. Furthermore, this approach restricts the types of customization that can be dynamically performed since the synthesized code must be anticipated by the kernel.

Solution (2.1) is supported by the Chorus micro-kernel. A Chorus actor defines a protected address space supporting the execution of threads which share the actor's resources [14]. A *supervisor actor* is both trusted and privileged, that is, it is allowed to perform sensitive kernel operations and shares the system's address space. From the perspective of efficiency, implementation of customized functions by means of supervisor actors is more or less equivalent to the code synthesis approach. Nevertheless, supervisor actors only permit static customization since the kernel must be regenerated each time a supervisor actor is added.

Solution (2.2) is safe since the application and the customized function execute in different address spaces. However, pointer conversion management can only be avoided by ensuring that entities visible in both tasks are mapped at the same virtual address in each task. Efficiency is more difficult to achieve since a task switch is necessary during communication and the called task may have to wait until several other tasks have been scheduled before it can be scheduled. Efficiency can be improved by using LRPC (lightweight remote procedure calls) [4] though execution in the called task is still penalized by the initial TLB (translation lookaside buffer) misses which result from having to invalidate the TLB on switching contexts.

Solution (3.1) is obviously unsafe since the user's application can modify data managed by the trusted customization code.

Solution (3.2) places the customized function in a private protection domain within the application's address space. Protection domains may be implemented either by software-based fault isolation [16] or by having the base operating system provide the means to create protection domains within a virtual address space. We do not consider the first approach due to its software orientation although it appears as a good candidate for a hardware-based fault isolation mechanism. Operating system support for protection domains can be achieved by a user level library which manages the page protections of the application's address space [3], altering them on a protection domain switch to match the visibility of the called domain. Alternatively, protection domain support can be integrated into the underlying micro-kernel ; such a solution is safe and inter-protection domain communication can be more efficient than LRPC since TLB entries need not always be invalidated on switching protection domains. Furthermore, this approach can also permit the dynamic creation of protection domains where threads execute with supervisor privilege. Specialized I/O functions [15], for example, could be run in such domains.

[1] Though we consider sharing of a virtual address space by multiple protection domains, we do not assume a single virtual address space operating system such as Opal [8] since the use of these operating systems for large scale distributed systems is still controversial (e.g., see [13]).

4 Towards Safe end Efficient Customization

Behavioral subtyping relations defined in the literature such as [2] and [11] rely on a *proof-theoretic* approach where the programmer is expected to reason in terms of specifications. For a system object implementing an operating system function, the object's specification may state properties related to the embedding system's state such as failures and objects' executing nodes. For instance, let us examine existing RPC systems: specifications of systems integrating support for replication [5] or atomic transactions [10] must in particular be given according to failures occurring within the underlying architecture; as another example, specifications of systems concentrating on high performance such as the one described in [4] must be given according to the calling and called executing nodes. We are currently examining appropriate specifications for system objects. Our first step towards this goal consists of specifying base system objects that give rise to numerous customizations; these include memory managers implementing various consistency models and existing RPC systems. We will then study the relationship between objects implementing the same system service according to behavioral subtyping, and in particular examine whether existing subtyping relations are suitable for our framework.

From the standpoint of operating system support for customization, we are currently examining the integration of protection domains within a multiple virtual address space operating system [6], our current target system being the Mach 3.0 micro-kernel. Our main design goal is that the cost of an inter-protection domain communication does not exceed the cost of an LRPC [4]. One use of protection domains is a *protected library*; this consists of executing a trusted, privileged function within the user's task in a protection domain distinct from that containing the application's code. The notion of protection domain we are investigating is actually more general; a protection domain may have no memory regions in common with other domains, it may share regions or it may completely englobe other domains.

This paper has discussed two complementary studies aimed at safe and efficient customization within distributed systems. This work is done in the framework of the Solidor project at IRISA/INRIA (Rennes, France) one of whose goals is to define programming and operating system support for easing management of large scale distributed systems.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for Unix development. In *Proceedings of the Usenix 1986 summer Conference*, pages 93–112, 1986.

[2] P. America. Designing an object-oriented programming language with behavioral subtyping. In *Foundations of Object-Oriented Languages, REX School/Workshop*, LNCS 489, pages 60–90. Springer-Verlag, 1991.

[3] A. W. Appel and K. Li. Virtual memory primitives for user programs. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 96–107, 1991.

[4] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 8(1), 1990.

[5] K.P. Birman. Replication and fault-tolerance in the ISIS system. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 79–86, 1985.

[6] C. Bryce and G. Muller. Implementing Efficient Protection Domains within Micro-Kernels. Technical report, IRISA - INRIA, 1994. In Preparation.

[7] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–516, 1985.

[8] J. S. Chase, H. M. Levy, M. J. Feeley, and E. D. Lazowska. Sharing and protection in a single address space system. *ACM Transactions on Computer Systems*, 12(3), 1994. To Appear, Also published as Technical Report 93-04-02, CSE Department, University of Washington, Seattle (WA, USA).

[9] M. B. Jones. Interposition Agents: Transparently interposing user code at system interface. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 80–93, 1993.

[10] B. Liskov. Distributed programming in Argus. *Communications of the ACM*, 31(3):300–312, 1988.

[11] B. Liskov and J. Wing. A new definition of the subtype relation. In *Proceedings of ECOOP'93*, pages 118–141, 1993.

[12] H. Massalin and C. Pu. Threads and input/output in the Synthesis kernel. In *Proceedings of the Twelfth ACM Symposium on Operating System Principles*, pages 191–201, 1989.

[13] S. J. Mullender and M. van der Valk. Integrating naming and addressing of persistent data in programming language and operating system contexts. In *Esprit Basic Research Project 6360,* BROADCAST *First year report, Vol. 2*, Newcastle, UK, 1993.

[14] M. Rozier. Chorus. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 39–69, 1992.

[15] C.A. Thekkath and H.M. Levy. Low-latency communication on high-speed networks. *ACM Transactions on Computer Systems*, 11(2):179–203, 1993.

[16] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 203–215, 1993.