Towards Operating System Support for Application-Specific Fault-Tolerance Protocols

Leonid Ryzhyk, Ihor Kuz

National ICT Australia* and University of New South Wales
Sydney, Australia
leonid.ryzhyk@nicta.com.au,ihor.kuz@nicta.com.au

Abstract

This paper proposes a new approach to operating system support for fault tolerance. We argue that in order to meet diverse application fault-tolerance requirements, the operating system should allow users to extend its functionality to support application-specific fault-tolerance protocols. We show that this kind of customisability can be achieved by explicitly decomposing the operating system into policies and mechanisms residing in different architectural layers and allowing applications to extend and modify these layers independently.

1 Introduction

Existing operating systems and middleware are designed on the assumption that the fault-tolerance needs of most applications can be covered by a small fixed set of fault-tolerance protocols. Systems built on this assumption provide inflexible fault-tolerance support that cannot be easily customised or extended. In particular, introduction of a new protocol that has not been envisaged during the design of the system is impossible without extensive changes to the core system structures and algorithms.

In contrast, we claim that fault tolerance is by nature *application-specific* — the choice of the right protocol in every particular case is determined by the structure and semantics of the application, its non-functional requirements, the types of hardware and software faults to be tolerated, and other factors. This claim is confirmed by the diversity of existing fault-tolerance protocols. Dozens of variations of checkpointing [CL85], transactions [BHG87], replication [FKL98], N-version programming [Avi85], recovery blocks [Ran95], software rejuvenation [KF95], and other techniques have been created to meet the needs of specific types of applications.

There is clearly no way a single operating system can implement all possible fault-tolerance protocols or even a reasonably large subset of those. This suggests that fault-tolerance strategies should be provided by applications themselves rather than by the operating system. On the other hand, the majority of fault-tolerance protocols require modifications to various aspects of the basic operating system functionality, including communication primitives, memory allocation, storage, and synchronisation, and thus cannot be implemented without operating system support. Combining these two observations, we conclude that a fault-tolerance-aware operating system should not directly implement any fault-tolerance protocols, but should rather be designed in an open way, which would allow users to introduce application-specific protocols by making the necessary changes to the operating system internal functionality.

To illustrate this requirement, consider the example of a cross-domain method invocation facility. The basic implementation of a method invocation that does not support any form of fault tolerance consists of marshalling invocation arguments, sending the invocation request to the server thread using blocking IPC, unmarshalling arguments on the server side, invoking the target object, preparing and sending the reply, and unmarshalling the returned value on the client side. Intro-

*National ICT Australia is funded by the Australian Government's Department of Communications, Information Technology, and the Arts and the Australian Research Council through Backing Australia's Ability and the ICT Research Centre of Excellence programs.

duction of fault-tolerance protocols into the system would require extending this sequence with additional steps. For instance, an object checkpointing protocol would involve logging and synchronisation of invocations, while transaction support would require appending a transaction ID to every invocation, checking the membership of the target object, and running a transaction concurrency control mechanism to acquire the necessary locks before performing the invocation. Other classes of fault-tolerance protocols would enforce their own changes to the communication mechanism.

In order to understand why existing operating systems do not support this kind of extensibility and what changes should be made to the operating system architecture to enable it, it is useful to logically decompose the system into *policies* and *mechanisms*. A mechanism defines a fragment of operating system functionality, while a policy defines how mechanisms are combined and used to provide a useful service to applications. For instance, some of the mechanisms involved in the above example include procedures for argument marshalling and unmarshalling, data transfer using synchronous IPC, message logging, and transaction concurrency control. A policy in this case defines the order in which these mechanisms are activated during the method invocation.

From the point of view of the differentiation between mechanisms and policies, most fault-tolerance protocols can be decomposed into mechanisms that must be added to the system and modifications that must be made to existing policies in order to make use of the new mechanisms. In the above example, invocation logging and synchronisation are mechanisms that must be provided to support the checkpointing protocol, while the corresponding policy that must be modified is the object invocation policy. While most existing operating systems can be easily extended with new mechanisms, there is usually no way to modify existing policies, because policy implementations are inseparable from the mechanisms they control. In our example, the cross-domain invocation policy is hardwired into remote invocation stubs together with all the associated mechanisms.

We believe that the lack of policy and mechanism separation is the fundamental cause of inflexible support for fault tolerance by existing operating system architectures. In the following section, we present a new operating system architecture that clearly separates policies and mechanisms into different architectural layers and allows application developers to introduce application-specific fault-tolerance protocols by redefining system policies on per-application basis.

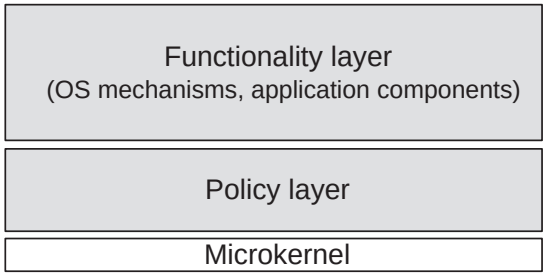

Figure 1: Architectural layers of the system.

2 Design sketch

The proposed operating system architecture consists of three layers (see Figure 1). The bottom layer is a small privileged microkernel providing basic mechanisms for hardware resource management and inter-process communication. The top layer is the *functionality layer*, containing all operating system mechanisms and application components. In between them lies the *policy layer* containing policies that control the structure and execution of the functionality layer. The policy layer defines what mechanisms and components reside in the functionality layer and how they connect to and interact with each other.

2.1 The functionality layer

The functionality layer is populated by two types of entities: operating system mechanisms and application components. While they play different roles in the system, they have the following common properties.

- **Indivisibility.** Functionality-layer entities are fundamental operating system building blocks. An entity can be added, replaced, or removed from the system as a whole, but not in parts. Therefore, different functions should be grouped into a single entity only if they are closely related to each other and are going to be used together in all conceivable scenarios.
- **Encapsulation.** A functionality-layer entity must provide well-defined functionality via a

well-defined access protocol. This property forms the basis for construction of complex operating system and application behaviours by composition of functionality-layer entities.

- **Policy freedom.** Functionality-layer entities must be free of policy decisions. A policy decision defines how different operations are combined together in a particular usage scenario. In other words, it defines how the functionality is used, as opposed to what it does. For instance, the decision to invoke the message transfer function right after marshalling method arguments during a remote method invocation is a policy decision — introduction of a fault-tolerance protocol into the system may require altering this decision in order to insert additional operations, e.g., message logging, in between these two operations. The goal of the system designer is to break the system functionality into policy-free units, so that policy decisions only need to be involved when crossing unit boundaries.

- **Communication only with the policy layer.** Functionality-layer entities are not allowed to communicate with each other directly. An entity can be executed only when being activated by the policy layer and can communicate with the rest of the world only by making invocations to the policy layer. Hence, the policy layer has complete control over the execution of the functionality layer. In traditional operating system parlance, the policy layer provides an operating system API to functionality-layer entities.

2.2 The policy layer

The policy layer consists of policies that define how functionality-layer entities are combined to provide more complex behaviours. Whenever a functionality-layer entity tries to interact with the outside world, e.g., to request an operating system service or to communicate with other entities, it invokes the corresponding policy. The policy defines a sequence of invocations to operating system mechanisms and/or application components that must be performed in order to complete the requested operation.

The internal structure of the policy layer is object-oriented. Policies associated with a single functionality-layer entity are grouped into a *policy object*. Methods of the policy object correspond to individual policies. Every functionality-layer entity, be it an application component or an operating system mechanism, is connected to a separate policy-layer object (see Figure 2). This design allows modifying policies associated with individual entities without impacting other entities.

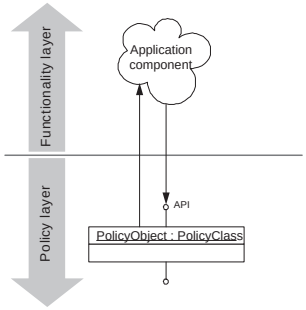

Figure 2: An application component and the associated policy object.

A policy object can communicate directly with all other policy objects in the system and with the functionality-layer entity connected to it, if there is one, but not with entities associated with other policy objects. Therefore, policy objects have to cooperate in order to invoke each other's functionality-layer entities. In order to eliminate the context switching overhead induced by the separation of policies and mechanisms, a policy object and its functionality-layer entity are always placed in the same protection domain.

3 Example

This section illustrates, by example, how the proposed architecture facilitates the use of application-specific fault-tolerance protocols. Consider an implementation of a hypothetical operating system API supporting a simple message-passing object model. Applications in this system are composed of objects residing within protection domains. Communication across protection domain boundaries is based on asynchronous message passing. For the purpose of this example, we will focus on a single aspect of the system — the implementation of the cross-domain message transfer facility. The corresponding policy-layer classes and interfaces are shown in Figure 3.

We start with a basic implementation, which does not support any form of fault tolerance. In Figure 4, object `Foo` sends a message to object `Bar`

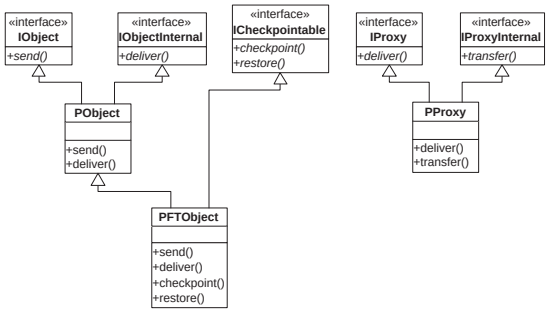

Figure 3: Policy-layer class diagram for the message-passing object model.

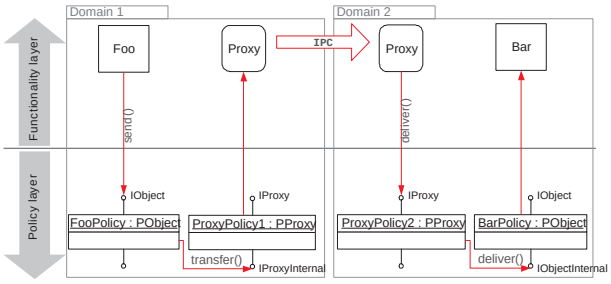

Figure 4: Cross-domain message transfer.

using the `send` API provided by its policy object. The communication is mediated by the message proxy mechanism that transports data across protection domain boundaries using the IPC primitive provided by the microkernel. The message transfer involves four policy invocations: the `send` policy of the `Foo` object, the `transfer` policy of the source message proxy, the `deliver` policy of the destination proxy, and, finally, the `deliver` policy of the `Bar` object.

The `PFTObject` class extends this basic implementation to support a simple checkpointing protocol based on pessimistic message logging [BBG83]. The checkpointing protocol works as follows. Before sending a message, an object saves the contents of all the messages it received so far into a stable storage. The contents of the log is regularly replaced with a copy of the complete object state. If one or more objects are destroyed as a result of a software or hardware fault, they can be restored by restoring the last checkpointed state of the object and replaying all the messages received after the checkpoint against it.

The `PFTObject` class redefines the `send` and `deliver` policies of the `PObject` class to perform message logging, as described above, and implements the `ICheckpointable` interface containing policies for checkpointing and restoring object state. The `checkpoint` policy writes the serialised representation of the object to the persistent store replacing the log of received messages, while the `deliver` policy restores the object from the latest checkpoint and then plays the messages from the log against it. Figure 5 shows the fault-tolerance protocol at work. Like in the previous case, `Foo` is sending a message to `Bar`. However, this time before transferring the message to the destination, `Foo`'s policy object writes the content of the volatile log to the non-volatile storage.

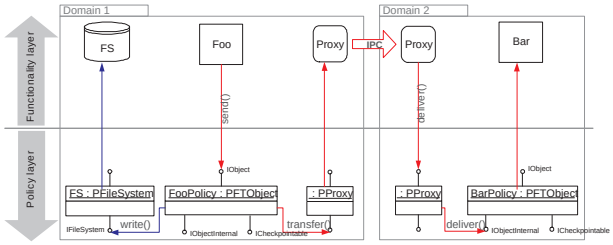

Figure 5: Fault-tolerant cross-domain message transfer

4 Conclusion

In this paper, we have argued that in order to meet diverse application fault-tolerance requirements, an operating system should allow users to extend its functionality to support application-specific fault-tolerance protocols. The paper has presented an operating system architecture that achieves this kind of extensibility by explicitly decomposing the system into policies and mechanisms residing in different architectural layers and allowing applications to redefine policies they use. The main advantages of the proposed architecture are as follows.

- It allows a wide range of fault-tolerance protocols to be introduced into the system by making relatively small incremental changes to the policy layer.
- The effect of the changes is restricted to the applications that use the given fault-tolerance protocols, while other applications can use different protocols or just continue relying on the default policies.
- The changes required to implement fault-tolerance protocols can be made in a well-structure object-oriented manner, using interfaces, inheritance, and method overloading.

References

[Avi85] Algirdas Avizienis. The n-version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, 11(12):1491–1501, 1985.

[BBG83] Anita Borg, Jim Baumback, and Sam Glazer. A message system supporting fault tolerance. In *Proceedings of the 9th ACM Symposium on OS Principles*, pages 110–118, October 1983.

[BHG87] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[CL85] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3:63–75, 1985.

[FKL98] Alan Fekete, M. Frans Kaashoek, and Nancy Lynch. Implementing sequentially consistent shared objects using broadcast and point-to-point communication. *Journal of the ACM*, 45(1):35–69, 1998.

[KF95] Nick Kolettis and N. Dudley Fulton. Software rejuvenation: Analysis, module and applications. In *Proceedings of the 25th IEEE International Symposium on Fault-Tolerant Computing*, page 381, 1995.

[Ran95] Brian Randell. The evolution of the recovery block concept. In Lyu, editor, *Software Fault Tolerance*, chapter 1, pages 1–21. 1995.