

DiPS: A Unifying Approach for developing System Software

Sam Michiels
Frank Matthijs
Dirk Walravens
Pierre Verbaeten

Report CW 332, February 2002



Katholieke Universiteit Leuven
Department of Computer Science

Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

DiPS: A Unifying Approach for developing System Software

Sam Michiels

Frank Matthijs

Dirk Walravens

Pierre Verbaeten

Report CW 332, February 2002

Department of Computer Science, K.U.Leuven

Abstract

In this paper we unify three essential features for flexible system software: a component oriented approach, self-adaptation and separation of concerns. We propose DiPS (Distrinet Protocol Stack), a component framework, which offers components, an anonymous interaction model and connectors to handle non-functional aspects such as concurrency. DiPS has effectively been used in industrial protocol stacks and device drivers.

Keywords : Self-adaptation, separation of concerns, component framework.

DiPS: A Unifying Approach for Developing System Software

Sam Michiels, Frank Matthijs, Dirk Walravens, Pierre Verbaeten
DistriNet, Department of Computer Science, K.U.Leuven
Celestijnenlaan 200A
B-3001 Leuven
Sam.Michiels@cs.kuleuven.ac.be

Abstract

In this paper we unify three essential features for flexible system software: a component oriented approach, self-adaptation and separation of concerns. We propose DiPS (Distrinet Protocol Stack), a component framework, which offers components, an anonymous interaction model and connectors to handle non-functional aspects such as concurrency. DiPS has effectively been used in industrial protocol stacks [7] and device drivers [10].

1 Introduction

This paper explains why component framework technology is needed for flexible system software (such as device drivers, protocol stacks and object request brokers) and how we are using DiPS (Distrinet Protocol Stack), a component framework, to build complex adaptable system software such as protocol stacks and device drivers.

We state that there are three essential features for flexible system software: a component oriented approach, self-adaptation and separation of concerns. There exist several systems and paradigms that each offer one or two of these features. Recent operating system research [4] [5] shows the advantage of providing reusable system components as basic building blocks. Other operating systems focus on the flexibility of the system that allows to adapt itself to changes in the application set it must support [13][12]. System software on tomorrow's embedded systems (such as personal digital assistants (PDA's), cellular phones or cars) must be intelligent enough to adapt the internal structure to new, even non-anticipated services and to integrate the necessary support for it (such as a new communication protocol or a new disk caching strategy), even at runtime. Strict separation of functional and non-functional code has proven to be an essential feature for adaptable, maintainable and reusable software [6][8][11]. In this paper we illustrate that the DiPS framework unifies component support, self-adaptability and separation of concerns in one paradigm, which is a strong combination for system software.

DiPS is not a complete operating system, but rather a component framework to build system software. We are convinced that other operating system abstractions, such as interrupt handling or memory management, can benefit from the DiPS approach. Our research is also heading in that direction.

We first introduce the main DiPS abstractions in section 2. We discuss a simplified disk driver design in section 3 to show the strengths of DiPS concerning component support, self-adaptation and separation of concerns. Section 4 presents some industrial projects that benefit from the DiPS component framework. The paper is summarized and some conclusions are given in section 5.

2 The DiPS approach

We now introduce the most important DiPS abstractions, such as `Message`, `Component`, `Connector` and `ReflectionPoint`. For an in depth discussion, we refer to [9].

2.1 DiPS Components

The coordination model (i.e. how components interact with each other) and the concurrency model (how to cope with concurrent threads) are completely separated from the functionality of the components. We will elaborate on the concurrency model in the next section when discussing DiPS connectors.

DiPS components are independent pieces of software which implement a well-defined functionality and offer a fixed interface. The modular approach allows to fine-tune software by collecting necessary and most suitable components (such as a disk scheduling, an error handling or a caching component). Components do not contain any internal multiprocessing to allow re-use in different concurrency contexts. A component interprets the information in an incoming DiPS `Message`, processes it and sends it further. This approach is similar to the tasks in Minix [15] which process messages. DiPS allows additional information to be attached to a message, which we call `meta-information`. This information can be anonymously provided by other components, or even by a user level service.

2.2 DiPS connectors

DiPS connectors are used to connect components. Such a separated coordination model (components do not know about each other) is necessary with dynamic self-adaptation in mind, since in that case restructuring, replacing, introducing or removing components is an essential feature.

Next to the separated coordination model, connectors also encapsulate the concurrency model. In traditional operating systems, such as Unix, synchronization code typically cross-cuts the functional code, which makes the system difficult to understand and maintain. By separating the concurrency model in the connectors, a well-trained system engineer can apply the concurrency model independent from the software builder who developed the functional components. Separation of concurrency code from the functional code is an important feature in other paradigms such as concurrent objects in [6] and aspects in [8].

Examples of DiPS connectors are the `MutexConnector` which only allows one thread at a time and the `ActiveConnector` which is composed of a buffer and a scheduler (see also the figure).

2.3 DiPS reflection points

Scheduling or cache replacement policies will often depend on the kind of services running on an operating system. Traditional desktop operating systems will often install a best-effort strategy which will have acceptable performance characteristics for most desktop applications. Disk intensive, memory consuming applications could perform better with a more specific disk scheduling or page replacement strategy. The modular DiPS approach allows the replacement of such a strategy.

DiPS reflection point connectors even allow self-adaptation by dynamically changing the communication path in a chain of components, instead of having to change components at run-time. A reflection point will decide what is the most suited forwarding path based on the meta-information in the incoming messages.

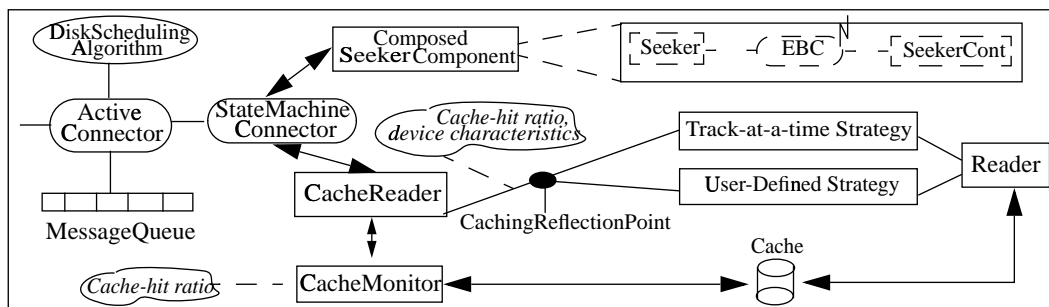
3 Example: a Disk Driver

In this section we explain how the DiPS approach benefits the development of device drivers. We propose a simplified disk driver only to show the advantages of the approach without overloading the discussion with

details. We therefore also make abstraction of the actual interrupt and error handling. We show the design of such a simplified driver in the figure. Connectors which just connect components, without extra concurrency support are not shown. The `Reader` component has been partially zoomed out; internally it has a similar structure as the `ComposedSeekerComponent`. The caching components (`CacheReader` and `CacheMonitor`) and the `CachingReflectionPoint` are logically part of the `Reader` component but are explicitly shown in the figure to simplify the discussion. We focus our attention on two aspects: separation of the typical wait/signal cycles between the driver and the disk controller from the functional code and self-adapting behavior according to caching strategies.

3.1 Separating coordination from functionality

Generally the task of a device driver is to translate an incoming request to one or more device specific requests, based on the status of the device and the type of request. When a hardware request is sent to a device controller, the driver is typically blocked at that point until the controller returns with an interrupt. (In some cases the hardware request is immediately returned, so that no blocking is needed.) For instance, when a read request arrives, the driver requests the disk controller to position the disk arm above a specific cylinder. This cannot be done immediately, so the driver is blocked until the disk arm is correctly positioned. This kind of coordination is often handled by inserting a sleep/wake up combination into the functional code.



DiPS completely separates this mechanism into an `EventBarrierConnector` (EBC) which acts as a barrier which is closed until both the message and an (interrupt) event are received. An example is shown in the figure. The `Seeker` component prepares a controller request to position the disk arm at a specific cylinder. The message is then blocked in the EBC until an interrupt event from the device is received. The `SeekerCont` component will interpret the answer from the disk after the barrier is opened. The management of state transitions is best separated from the functional components [2]. Therefore we introduce a separate `StateMachineConnector` which manages the internal state machine of the driver and delegates messages to different components, based on the current state of the driver. By separating the status management from the functionality, components can easily be reused in other contexts.

3.2 Self-adapting behavior

Suppose we provide the disk driver with a data cache and some caching strategy such as a `Track-at-a-time Strategy` (see the figure) [15] to improve I/O performance. By using such a strategy, a whole track is read instead of one sector. The idea is that much more time is lost by seeking a specific cylinder compared to the rotation or transfer time. So, once the disk arm is moved to a specific cylinder, it hardly matters whether one sector is read or the whole track. Several read requests that are closely together on the disk (for example reading a large contiguous file) will result in a higher cache-hit ratio (by the `CacheReader`) and as a consequence performance is gained.

The performance gain of a track-at-a-time strategy is lost though when the physical data blocks of a file are widely scattered on the disk, because cache-hit ratio will be poor in this case. It would be better to dynamically choose for a more appropriate caching strategy (cfr. the `User-defined Strategy` in the figure). Note that some disk controllers do hardware caching. When such a hardware caching device is added to the

system there is no point in having the driver do it as well (for that specific device). This clearly shows the need for dynamic self-adaptation, driven by user level services and the system itself.

The component approach in DiPS allows that user-level services customize system behavior by integrating new, even non-anticipated, policy components [3]. To achieve self-monitoring behavior we should introduce monitoring components in the driver itself (for example the `CacheMonitor` which provides `cache-hit ratio` info) or in other subsystems such as the file system (cfr. `device characteristics` info). This information flows through the system as meta-information, which is attached to DiPS messages and is used by the `CachingReflectionPoint` to decide which is the most appropriate caching strategy of the moment to delegate incoming messages to.

4 Industrial context

We now discuss two industrial projects our research group is involved in. Since these projects have extended flexibility requirements, they heavily benefit from using the DiPS component framework. The first project focuses on protocol stacks, the second on USB device drivers.

4.1 A flexible protocol stack environment

The context of this project are vehicles which are able to communicate wireless with a base centre via several communication channels (and vice versa) [7]. Services (agents, applications) can be dynamically uploaded to the vehicle. Which communication channel is used for interaction between a service on a vehicle and the base centre is completely transparent. Dependent on the services that are available on the vehicle, different communication protocols are needed. To send e-mails, a regular TCP/IP protocol stack is needed, for voice-over-IP, extra protocols are necessary and so on. It is clear that the system on a vehicle cannot support all possible services which will be invented in the near future, nor is it acceptable that the vehicle software must be reinstalled every time a user subscribes for a service which requires a new communication protocol... [13]

The DiPS component approach allows to insert new communication layers into a running protocol stack, which is not a trivial challenge. It also allows user level services to influence the underlying system by expressing communication preferences (such as communication speed or cost). These preferences flow through the protocol stack as meta-information attached to each network packet (message). This allows to switch between communication channels whenever necessary (for instance when GSM coverage is lost during data transmission). A specialized composition algorithm is used to translate service specific requirements into a stack of communication protocols and hardware drivers [14].

4.2 A flexible device driver environment

This project is focused on testing devices, i.e. checking whether the operation of a device completely follows the specification of its standard (in casu USB [1]). Complex driver architectures, such as USB, have a layered architecture. Testing a device's USB functionality in all situations requires in the first place a device driver that allows test applications to inject specific message sequences, manipulated data and even malformed control messages at any place in the driver stack to see whether the device reacts properly. A modular and an open device driver implementation is essential to allow such tests. Today's USB implementations do not allow such a flexible manipulation.

On the other hand it is crucial that changes in the protocol specification itself can easily be integrated in a driver implementation by adding, replacing or removing basic building blocks. The USB standard is a rapidly evolving new standard that is likely to change.

The DiPS approach is very effective to develop such flexible and complex software as a USB driver architecture [10]. Relevant contributions of DiPS include its anonymous communication model (meta-information) and connectors for handling non-functional issues such as the interaction and the concurrency model. We are

convinced that DiPS is well-suited not only for USB drivers, but for flexible device driver development in general [10].

5 Conclusions

In this paper, we show the advantages of using DiPS, a component framework, to develop system software, such as protocol stacks and device drivers. Projects with industrial partners have clearly shown the need for adaptability of system software to user level requirements.

DiPS unifies powerful paradigms, such as component oriented programming, self-adaptation and separation of concerns. This strong combination allows us to develop a flexible protocol stack for wireless vehicle communication and a USB device driver architecture.

Experiences in applying DiPS in these two system software areas drive our research towards other operating system abstractions, such as interrupt handling, CPU scheduling and memory management.

6 References

- [1] Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, NEC and Philips. Universal serial bus (USB) 2.0 specification. <http://www.usb.org/>, Apr. 2000.
- [2] Design Patterns. E. Gamma, R. Helm, R. Johnson, J. Vlissides. Addison-Wesley, 1995.
- [3] D. Devriendt. A dynamically changing web server. M.Sc. Thesis, K.U.Leuven, Department of Computer Science, 2000.
- [4] B. Ford, K. Van Maren, J. Lepreau, e.a.. The FLux OS toolkit: Reusable Components for OS implementation. Proceedings of the Sixth IEEE Workshop on Hot Topics in Operating Systems, May 1997.
- [5] E. Gabber, C. Small, J. Bruno, J. Brustoloni, and A. Silberschatz. The Pebble Component-Based Operating System. In Proc. of the USENIX 1999 Annual Technical Conf., June 1999.
- [6] J. Itoh, Y. Yokote and M. Tokoro. Scone: Using concurrent objects for low-level operating systems programming. Technical report, Department of Computer Science, Keio University, 1995.
- [7] P. Kenens, S. Michiels, W. Joosen, F. Matthijs, P. Verbaeten, e.a.. The SmartMove communication architecture. Technical report, SmartMove internal use only, K.U.Leuven, Aug. 1999.
- [8] G. Kiczales, e.a. Aspect-Oriented Programming. Proceedings of ECOOP'97, Finland. Springer-Verlag LNCS 1241, pp. 220-241. June 1997.
- [9] F. Matthijs. Component Framework Technology for Protocol Stacks. PhD thesis, K.U.Leuven, Department of Computer Science, Dec. 1999.
- [10] S. Michiels, P. Kenens, F. Matthijs, D. Walravens, Y. Berbers and P. Verbaeten. Component Framework Support for Developing Device Drivers. Proceedings of International Conference on Software, Telecommunications and Computer Networks, vol. 1, FESB, Split, Croatia, pp. 117-126. June 2000.
- [11] B. Robben. Language Technology and Metalevel Architectures for Distributed Objects. PhD Thesis, Department of Computer Science, K.U.Leuven, May 1999.
- [12] M. I. Seltzer, e.a.. An Introduction to the architecture of the VINO Kernel. Harvard University Center for Research in Computing Technology Technical Report TR-34-94, 1994.
- [13] M. Seltzer and C. Small. Self-monitoring and Self-adapting Operating Systems, Proceedings of the Sixth Workshop on Hot Topics in Operating Systems, May 1997.
- [14] I. Sora, S. Michiels, F. Matthijs, D. Walravens, e.a. Policies for Dynamic Stack Composition. Internal Technical Report, Department of Computer Science, K.U.Leuven, Nov. 2000.
- [15] A. Tanenbaum. Operating Systems, Design and Implementation. Prentice-Hall. 1987.