COMPONENT FRAMEWORK SUPPORT FOR DEVELOPING DEVICE DRIVERS

SAM MICHIELS, PETER KENENS, FRANK MATTHIJS,
DIRK WALRAVENS, YOLANDE BERBERS, 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 advocate the use of component framework technology for developing state-of-the-art system software. Relevant contributions of DiPS (Distrinet Protocol Stack) [Mat99], a component framework, include its anonymous interaction model, connectors for handling non-functional issues such as the concurrency model, and builder support. DiPS has effectively been used in industrial protocol stacks [KMD+99] [Mat99]. This paper shows how we are using the DiPS component framework to build and support flexible device driver software.

1 Introduction

This paper explains why component framework technology is needed for flexible system software (such as device drivers, protocol stacks or object request brokers) and how we are using DiPS (Distrinet Protocol Stack) [Mat99], a component framework, to build and support flexible device driver software. Flexible system software allows user- or application- specific preferences to be reflected in the underlying (operating) system such that system behavior can be adapted if necessary.

The DiPS framework has originally been built for protocol stack development (which explains the name). The framework approach allows reuse of basic components resulting in rapid product development. The component approach also supports dynamic system behavior, which has become essential for modern system software where applications, devices and the operating circumstances change continuously.

The DiPS component framework has been effectively used in industrial protocol stacks [KMD+99] [Mat99]. Current use of the DiPS framework

for device driver and object request broker development shows that it is extremely effective for other system software as well. We are convinced that run-time flexibility in device driver software is needed.

Section 2 shortly describes two cases showing the need for dynamic flexibility in device driver software. Section 3 explains our view on framework and component technology and the reason why component and framework technology are a strong combination for developing dynamically adaptable device drivers. In section 4 we describe the most important abstractions of the DiPS component framework and its major contributions. Section 5 shows how DiPS has been used to develop flexible device driver software. The paper is summarized and conclusions are given in section 6.

2 Flexibility in Device Drivers

It is hard to configure device drivers that behave optimally in all conditions. Scheduling or caching strategies are often specialized towards specific situations and cannot adapt to unusual circumstances. Research in device driver development shows that device driver software needs support for concurrency control and adaptive behavior [IYT95].

2.1 Ad-hoc network drivers

Devices become more and more intelligent. An example of such an 'intelligent' technology is Bluetooth [Gro99]. Bluetooth devices are connected in an ad-hoc fashion, i.e. not requiring predefinition and planning, as with a standard network.

The flexibility offered by Bluetooth must be adaptable to changing circumstances, such as the battery level of a device. For example, a low-battery Bluetooth-enabled mobile phone won't allow nearby devices to route packets to the Internet (while it would do routing under better battery conditions). The only way to enforce that is to provide support for it in the Bluetooth driver.

System software must be adaptable to support a variety of Bluetooth services, i.e. different protocols and drivers are necessary for different services [Met99]. It is clear that protocols and drivers are only needed for those services that are used, and only for the time that they are actually in use. Consequently, the integration (and removal) of system software at run-time must be supported.

Static flexibility is not effective enough in such a dynamic Bluetooth environment. The behavior of a Bluetooth system is likely to be different

depending on circumstances such as the running application, the user of the device, its current power status, the nearby devices or a combination of all these. It is therefore clear that there is a need to dynamically monitor the system and to appropriately anticipate to changing circumstances at run-time.

2.2 USB hub drivers

In bus architectures such as USB [CHPI+00], devices can be added or removed at run-time. This means that the associated device drivers have to be loaded and unloaded as well. Key nodes in such a dynamic architecture are the hub drivers. They are responsible for dynamically building and managing the topology of devices that are connected to a hub.

The traditional USB architecture provides basic functionality to load and unload a corresponding driver when a device is added or removed. Dynamic control of which drivers are allowed to be loaded is clearly not supported. It should be possible to enforce an algorithm that decides which drivers are allowed to be loaded under certain circumstances. A useful algorithm prevents speaker devices from being activated at a clerk's desktop computer from 8 am till 5 pm... Another example is preventing a floppy drive to be used or activated without permission to protect a workstation from being infected with a virus through an infected floppy disk.

Static policies cannot offer sufficient flexibility. For example, depending on the user that logs in at the system, different policies must be applied. It is clear that dynamic flexibility is needed to support this.

3 Component Framework support for Device Drivers

Framework technology is a well-known technique for building complex software systems and has been successfully used in system software [CIMR]. In this section we will describe why traditional frameworks cannot offer support for dynamic flexibility. We therefore define our view on component and framework technology. We will describe how a component framework will help in developing dynamically adaptable system software.

3.1 Flexibility at design-time: Frameworks

The idea behind frameworks is that, for a specific problem domain (such as graphical user interfaces, networking, banking applications, secure transactions, broadcast media automation, etc.), there are significant similarities between different applications within a single domain.

The following reasonably established definition covers this idea: "a framework is an abstract design for a particular kind of application" [JF88].

As for the structure of frameworks, there are two extreme approaches: *white-box* frameworks and *black-box* frameworks. The former kind is based on a large number of abstract classes from which can be inherited. The latter kind of framework structure is not based on inheritance, but on object interactions with well defined interfaces.

Frameworks are design-time artifacts and are not explicitly present anymore at run-time. This way, when the system is running, we lose the flexibility we had at design-time (e.g. easily integrating new functionality, replacing existing functionality, etc.). To gain this flexibility also at run-time, a component framework is mandatory.

3.2 Components

Component-oriented programming may be seen as the next logical step in object-oriented software development, whose main purpose is to support large-scale re-usability. A component can be defined as a unit of composition with contractually specified interfaces and explicit context dependencies. Components can be deployed independently, and are subject to composition by third parties [Szy98].

It is important to note that components are not just design entities but actual products that live at run-time. Open systems need to deal with run-time issues such as coordinating concurrent access to shared components or other resources, establishing a certain order of interactions, selecting which component will receive a certain request, etc. These concerns are operational rather than functional.

In the open market model of component-oriented programming, it is recognized that free composition does not allow very well to systematically cope with operational characteristics. The current idea in component-oriented programming is that, in order to be able to deal with system-wide issues, some restrictions are needed in the way components are connected. These restrictions are enforced by what is called a *component framework*.

3.3 Flexibility at run-time: Component Frameworks

Component frameworks essentially enforce a restricted composition model. A component framework contains and explicitly defines (part of) the connectors which connect components in a component-based software system. It is explicitly present at run-time to enforce certain global properties. The component framework needs specific information from the components which are

plugged in, which adds restrictions to the components that can successfully be used together with the component framework.

4 The DiPS Component Framework

We have used component framework technology in DiPS to capture the domain of system software (such as protocol stacks, device drivers and object request brokers). In this section we describe the most important abstractions of DiPS (*packet*, *component* and *connector*).

Next to the basic support for components and connectors, DiPS offers domain specific support for building system software based on a high level of abstraction. Other DiPS contributions, such as the anonymous composition model and support for builders are handled in section 5. For a detailed description of DiPS we refer to [Mat99].

4.1 Packet

The information that flows through a system is encapsulated in a so-called packet. A packet is more than just a container of data though. It encapsulates the characteristics of the data or the application that is sending the data. We call this extra information *meta-information*. The use of meta-information supports anonymous communication throughout the system. One component produces some information, others can independently consume it.

4.2 Component

Packets are passed to components via a fixed interface. Each component interprets the information contained in the packet, processes it and sends it further. DiPS offers composite components which are built by aggregating basic components into larger ones. Using fine-grained components as basic building blocks has many advantages such as modularity, re-usability of components and high selection precision. However, coarse-grained building blocks are easier to handle than fine-grained ones. DiPS allows developers to choose the granularity level of their components.

Components are purely passive. By sticking to passive components, the framework is able to separate the functionality in the system from the coordination model. The coordination is handled by the connectors which will be described next. The same component can therefore be used synchronously or asynchronously, depending on the context (e.g., the other components) in which it operates.

4.3 Connectors, concurrency support and customization

The third key abstraction of the framework is the *connector*. A connector is a programmable entity which is responsible for the interaction between components. Components in the DiPS framework never interact directly with each other. The interaction is always made through connector objects. In fact, a component does not even know to which other components it is connected (if any). This is the responsibility of the connectors. The component that is connected to the output of a connector is called the *destination component*. The component(s) connected to the input is (are) called the *source component(s)*.

Connectors are also concerned with operational issues, such as the concurrency model. Because DiPS clearly separates the concurrency model from the functionality, the concurrency model can be dynamically adapted by replacing connectors. The DiPS component framework offers pre-defined connectors which support different concurrency models. In addition, new connectors can be developed and individually reused. An example of such a connector is the *mutex connector* which simply connects the output of one component with the input of another component with additional mutual exclusion semantics. Another frequently used connector is the *active connector*. This connector is composed of a buffer and a scheduler. The scheduler selects packets from the buffer and sends them to the destination component.

DiPS provides a special connector for customization, namely the *reflection point* connector. A reflection point has one input and many outputs. It observes the traffic that passes through by interpreting the meta-information encapsulated in a packet. Based on these observations, it makes a decision about how to route the traffic through the system and it selects one of its outputs according to that decision.

5 DiPS support for Device Drivers

This section describes how DiPS has been used to support the development of adaptable device drivers. First we describe how DiPS contributes to the development of device drivers and how it offers support for flexibility at design-time as well as at run-time by its anonymous composition model and its builder support. Secondly, we apply the DiPS approach to develop a flexible USB hub driver.

5.1 DiPS contributions for Device Drivers

Components. Device driver functionality is composed of components such as dynamic device detection, interrupt handling, caching and scheduling. Components are (re)used for different purposes. For example, a recognizer and a loader component can be used for dynamic device detection: a recognizer component to discover the configuration of a device and a loader component to load the appropriate driver into memory.

Connectors. Connector support is essential in a concurrent device driver environment, for instance to handle hardware interrupts or to enforce mutual exclusion. A hardware signal is transformed to a DiPS packet at the lowest software layer to handle an interrupt in the DiPS component framework. An interrupt packet is sent to a DiPS active connector, where it is buffered in case the destination component cannot handle the packet immediately. When the destination is ready, the active connector schedules the next packet and sends it to the destination.

Another very essential connector is the mutex connector. This connector enforces that packets are processed sequentially by the destination, without the need for synchronization code in the component. Instead of being confronted with this difficult and error-prone aspect, programmers can concentrate on the pure functionality of the component they are developing.

Anonymous component composition. Components are anonymously composed by connectors. As a result, each individual component can be fully reused since it has no explicit references to other components. A second advantage is that, thanks to the anonymous interaction model, the DiPS component framework supports open environments where components and connectors can be (dynamically or statically) added or removed. Other components or connectors do not have to know about these changes, and do not need to obtain a reference to the newly introduced entities.

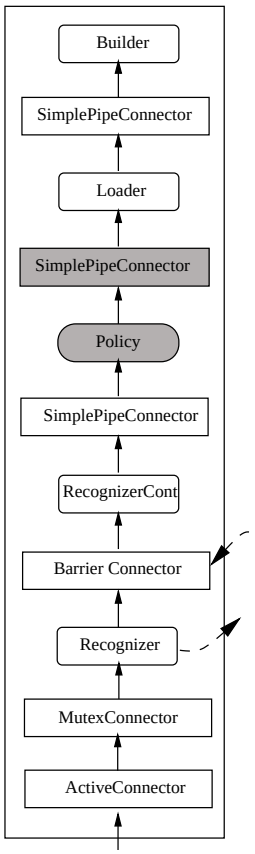
The anonymous composition model is essential in developing prototype device drivers. Initial prototypes only offer the most essential functionality. General components are reused to offer functionality such as buffering or a basic scheduling algorithm. Incrementally, the prototype is extended by replacing components with more specific implementations or by integrating new components that provide more functionality.

Builder support. The DiPS support for builders allows a programmer to think about building software in terms of connecting components and connectors. DiPS builders interpret a program written in a descriptive lan-

guage (eXtensible Markup Language [GQ99]). A builder allows to build or re-structure software statically or dynamically.

In section 2 we showed that dynamic adaptation support for device drivers is necessary. A policy component that influences the behavior of the system (for example user specific behavior) can easily be integrated just as any other component. Note however that statically built software systems can sometimes be sufficient to fulfill the customization needs. DiPS offers both approaches.

Replacing state-full connectors at run-time (for example an active connector with a non-empty buffer) must be done with care, to prevent loss of information. A builder therefore keeps both connectors until the old connector is ready to be removed. Meanwhile, new requests are delegated to the new connector. It should be clear that this problem does not arise with design-time restructuring.



5.2 Applying DiPS

We did apply the DiPS abstractions that are described previously in a high-level design of a flexible USB hub driver. This design is shown in the figure. We omit the hardware abstraction layer and the rest of the USB architecture. We will explain how the DiPS component framework offers support for concurrency in a device driver and how the behavior of a device driver can be adapted by introducing new components and connectors. The main responsibilities of a USB hub driver are to detect when a device is plugged in or out of a hub device and to make sure that the corresponding device driver is loaded or unloaded. We only describe the attachment of a new device to a hub, the removal case is similar.

Hardware signals are converted to DiPS packets at the hardware abstraction layer. Those packets are buffered in an `active` connector when they cannot be handled immediately. A `mutex` connector enforces that only one packet at a time is handled. The device recognition process is actually divided into two components. The `recognizer` component is used to send a device identification request as an event. This event is handled by other components in the USB system which are able to discover the configuration of the device. When the recognition process is finished, a reply event is sent which is received by the `barrier` connector. The `recognizer cont` component, which acts as a continuation of the recognition process, is visited after

the `barrier` connector has released the packet. This connector synchronizes the two recognition components by buffering an incoming packet until a specific event is received. Traditional systems, such as UNIX, offer the *sleep* and *wakeup* mechanism which is too complex and not modular. DiPS offers a clear separation of functional code in the components and synchronization code in the connectors. A similar approach is defined in [IYT95].

When the device is recognized, a `loader` component locates the driver and actually loads it into memory. A specialized driver `builder` component finally instantiates the driver and connects it in the USB system structure. The builder approach is used at two different levels in the hub driver. On the one hand, a device driver builder is used to compose a hub driver by connecting components and connectors. On the other hand, an internal `builder` component is used to dynamically integrate new driver components in the system when a device is connected to the hub.

Adding more intelligence to the hub driver is a matter of introducing new components. We introduce the `policy` component that we described in section 2 (see the dark shaded boxes in the figure). This component can decide whether a device driver is allowed to be loaded or not. It is reasonable to place the `policy` component between the recognition process and the `loader`. We must have recognized the device which is connected, but we want to enforce the policy before the driver is actually loaded into memory.

6 Conclusions

In this paper, we advocate the use of component framework technology for developing state-of-the-art system software. Although object-oriented frameworks have been successfully applied for building system software, many current technologies (such as Bluetooth and USB) are very dynamic in nature and can not be supported effectively by such static, design-time artifacts. We strongly believe in the strength of combining framework and component technology for offering the required flexibility, both at design time and at run-time.

Our observations are based on experience with the DiPS component framework, which has been built according to this view. While DiPS has been developed to support protocol stack development, in this paper we show the strengths of DiPS for creating flexible device driver software. Relevant contributions of DiPS include its anonymous interaction model, connectors for handling non-functional issues such as the concurrency model, and builder support.

This paper shows how these properties are extremely effective in supporting flexible device drivers. Advantages include incremental driver development, introduction of new driver components (such as policies) both at design-time and at run-time, and separation of concerns, which leads to increased reuse, easier to understand driver structure and faster and less error-prone driver development.

We applied DiPS in a USB device driver prototype and are currently working on versions with different concurrency support.

References

[CHPI+00] Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, NEC, and Philips. Universal serial bus (usb) 2.0 specification. http://www.usb.org/, April 2000.

[CIMR] R. Campbell, N. Islam, P. Madany, and D. Raila. Implementing Choices: an object-oriented system in C++.

[GQ99] I. S. Graham and L. Quin. *XML Specification Guide.* Wiley Computer Publishing, 1999.

[Gro99] Bluetooth Special Interest Group. Bluetooth core specification v1.0b. http://www.bluetooth.com/, December 1999.

[IYT95] J. Itoh, Y. Yokote, and M. Tokoro. Scone: Using concurrent objects for low-level operating system programming. Technical report, Department of Computer Science, Keio University, 1995.

[JF88] R. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, June 1988.

[KMD+99] P. Kenens, S. Michiels, O. Debels, S. Van Baelen, W. Joosen, F. Matthijs, and P. Verbaeten. The SmartMove communication architecture. Technical report, SmartMove internal use only, K.U.Leuven, August 1999.

[Mat99] F. Matthijs. *Component Framework Technology for Protocol Stacks.* PhD thesis, K.U.Leuven, Department of Computer Science, December 1999.

[Met99] R. Mettala. Bluetooth protocol architecture version 1.0. White paper, August 1999.

[Szy98] C. Szyperski. *Component software: beyond object-oriented programming.* ACM New York, 1998.