Applying Component Frameworks to Develop Flexible Middleware

Nikos Parlavantzas, Geoff Coulson and Gordon S. Blair

Distributed Multimedia Research Group,
Computing Department, Lancaster University,
Bailrigg, Lancaster LA1 4YR, UK.
{parlavan, geoff, gordon}@comp.lancs.ac.uk

1. Introduction

It is now well established that middleware must accommodate a wide variety of requirements imposed by applications and underlying environments. Moreover, it must be able to absorb both static and dynamic changes in those requirements. The current generation of middleware clearly fails to address this need [1]. To help address such concerns, we believe that middleware should be built according to a component-based architecture by wiring together independently developed components. It is perhaps ironic that while current middleware intends to support the compositional paradigm for building applications, it itself suffers from an inflexible and monolithic approach to development.

The benefits of component technology are well known: reusability, extensibility, modularity, understandability, and reduced development costs. However, while component technology *permits* runtime adaptability and extensibility, it does not by itself *guarantee* such levels of flexibility. Our solution to this is the use of reflection and open implementation, whereby aspects of middleware behaviour are reified and made accessible at runtime for inspection and adaptation through a *meta-interface* (these aspects can be naturally encapsulated as component instances). However, designing these meta-interfaces can be hard! The 'obvious' solution of exposing the whole middleware implementation as a graph of component instances and allowing arbitrary manipulations is clearly not acceptable for reasons of robustness; the problem of maintaining consistency in the face of dynamic adaptation is critical.

Our current approach to addressing these problems is through the notion of *component frameworks*. In this position paper, we discuss the role of component frameworks within componentized middleware, outline a proposed architecture, and conclude with some open questions and discussion topics.

2. The role of component frameworks

A component framework (CF) is a "collection of rules and contracts that govern the interaction of components plugged into it" [2]. CFs are primarily design-level entities. However, in our architecture they are themselves realised as components that enforce some of these rules by implementing common mechanisms and regulating interactions. The primary motivation for CFs is to provide built-in architectural properties and invariants by *constraining* the design space of components. Moreover, CFs simplify component development and assembly and increase the understandability and maintainability of the system. They typically address a specific and focused problem domain (e.g., implementing communications protocols), and thus many CFs may need to be integrated in a component system.

Note that the CF concept is distinct from OO frameworks in the sense that CFs are not bound to a specific programming language, and there is no implementation inheritance between components and framework. As a result, components and CFs can be distributed in binary form, be independently developed, be combined at runtime, and evolve independently from each other.

In the context of our work, CFs have the added benefit of offering a convenient 'hook' to which robust and meaningful meta-interfaces can be attached. Through these meta-interfaces, the implementation of CF-based (sub)systems (i.e., configurations of instances obeying the CF rules) is exposed in a controlled way. The meta-interface is designed as part of the CF and is usually presented in terms of domain-specific abstractions. Importantly, the meta-interface can exploit the domain knowledge built into the CF in order to enforce a desired level of (domain-specific) consistency. The degree of flexibility afforded by the meta-interface is also determined by the CF design and has to be balanced against concerns for efficiency, assured consistency, and understandability. Typically, the meta-interface is implemented by a CF-provided component that maintains information about the current configuration of embedded component instances and applies it to perform adaptations.

3. Towards a component-based architecture for middleware

At the lowest level, we adopt a component model that defines a binary interfacing standard (in our case, the COM binary standard) allowing components to be dynamically loaded and combined within a single process. We then provide a minimal support infrastructure that offers basic services for finding, loading and instantiating components. Other services that traditionally belong to middleware (e.g., remote method invocation) are then implemented on top of this minimal infrastructure either as simple components or as CFs. Similarly, applications are built from components that follow the same component model (but typically belong to different CFs). The use of a uniform component model is proving highly beneficial in terms of development cost, interoperability, and portability of components.

Essentially, the middleware architecture can be viewed as a set of interoperating component frameworks. We are currently employing CFs for binding establishment and control, pluggable protocols, and thread and buffer management, each featuring a CF-specific meta-interface. We have found that the use of multiple CFs provides a useful separation of concerns and a controlled scope for adaptation and extension. It also frees the designer to choose the particular architectural style and meta-interface that seems appropriate for each problem domain. For instance, a CF for building communication protocols may benefit from an event-based architectural style with data sharing (like Coyote), whereas a CF for multimedia streaming functionality may employ a pipes-and-filters style (Microsoft DirectShow, Open bindings). In the first case, a possible meta-interface may enable the rebinding of events to different event handlers, while in the second case it may enable the reconfiguration of the filter graph.

4. Discussion

As our research is at a preliminary stage (we are currently implementing CFs for scheduling policies and pluggable protocols), we would like to raise and discuss a number of issues at the workshop, including:

(1) What is the minimal infrastructure necessary to support a suitable component model? For example, there are proposals that advocate maintaining static and dynamic dependencies of components in the infrastructure [3]. Is this required and what is the overhead?

(2) What component frameworks are useful and necessary in a reflective middleware platform?

(3) What guidelines and styles are useful for designing flexible and dynamically adaptable component frameworks with robust meta-interfaces? Similarly for designing the framework for interoperation between CFs. Do we need "higher-order" component frameworks as proposed in [2] and what would they look like?

(4) How does one deal with components that have to be plugged in more than one CFs? Do we actually need that or do we forbid it?

(5) Do we need to explicitly support the evolution of a CF at runtime and how? It seems that basic component mechanisms (multiple interfaces, runtime discovery, versioning, etc) provide a partial solution to that. However, one must be careful to preserve compatibility with both existing embedded components and with other components and CFs that make use of it.

References

[1] Gordon Blair, Geoff Coulson, Philippe Robin and Michael Papathomas, "An Architecture for Next Generation Middleware". Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing: Middleware'98, The Lake District, U.K., 15-18 September 1998, pp191-206.

[2] C. Szyperski, "Component Software. Beyond Object-Oriented Programming", Addison Wesley, ISBN: 0-201-17888-5, 1997.

[3] Fabio Kon, Manuel Román, Ping Liu, Jina Mao, Tomonori Yamane, Luiz Claudio Magalhães, and Roy H. Campbell, "Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB". IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000). New York. April 3-7, 2000.