Reflections on Aspects and Configurable Protocols

Matti Hiltunen
AT&T Labs - Research
Florham Park, NJ 07932, USA
hiltunen@att.com

François Taïani
Computing Department
Lancaster University
Lancaster LA1 4WA, UK
taiani@comp.lancs.ac.uk

Richard Schlichting
AT&T Labs - Research
Florham Park, NJ 07932, USA
rick@research.att.com

ABSTRACT

The goals of aspect oriented software development (AOSD) and frameworks for configurable protocols (CPs) are similar in many respects. AOSD allows the specification of cross-cutting concerns called aspects as separate modules that are woven with the base program as needed. CPs are oriented towards building protocols or services with different quality of service (QoS) properties and attributes out of collections of independent modules, with each configuration customizing the service for a given application and execution environment. As AOSD evolves to address issues in areas such as middleware, operating systems, and distributed computing that have traditionally been the domain of CPs, lessons learned from the development of these frameworks could be useful. The purpose of this paper is to draw parallels between AOSD and CP frameworks, with a specific focus on the Cactus framework and how it compares and contrasts with the aspect-oriented paradigm.

Categories and Subject Descriptors

D.3.3 [**Language Constructs and Features**]: Frameworks; D.1.0 [**Programming Techniques**]: General

Keywords

Configurable software, extensible software

1. INTRODUCTION

The need to deal with non-functional requirements has long been recognized as a factor that increases the complexity of software. These requirements include broad system attributes such as fault tolerance, timeliness, and security—*quality of service (QoS) attributes*—as well as more targeted programming issues such as exception handling, memory layout, and debugging. While these requirements span a broad range of concerns and occur in a wide variety of contexts from single machines to large distributed systems, they are all unified by the fact that their realization requires code beyond that needed to implement the base functionality of the software. As such, non-functional requirements often complicate the software by imposing "nonlinear" control flow that is difficult to implement cleanly in normal procedural programming models. Research related to this issue has been the focus of a number of diverse communities and efforts, including reflection [44], aspect oriented software development (AOSD) [28], and configurable network protocols and distributed services (CP) [4, 22].

The goal of this paper is to draw parallels between the approaches developed for AOSD and CP, with a special focus on relating lessons learned from the development and use of the Cactus framework [4, 24] that are relevant for AOSD. Cactus is one of a number of systems that have been developed over the years to support highly modular implementations of network subsystems, individual network protocols, and services in distributed systems. Many of the early systems such as System V STREAMS [40] and the *x*-kernel [26] have composition and execution models that are oriented towards hierarchical or linear combinations of protocol modules, i.e., network stacks. However, while sufficient for traditional layered network subsystems, the constraints imposed by linear execution make it difficult to use these systems for higher-level protocols and services in which non-functional requirements are more prominent. As one example, for a data transport service, these requirements may include message reliability, preservation of message ordering, flow and congestion control, timeliness properties such as bounded maximum latency or jitter, and security properties such as confidentiality, integrity, and non-repudiation. In much the same way as with AOSD, the cross-cutting nature of these requirements argues for new models and new programming paradigms.

The Cactus composition and execution model is designed to address issues related to non-functional requirements by supporting fine-grain non-hierarchical composition and event-driven execution. In Cactus, a *service* such as group communication or QoS enhancements for distributed object systems [18] is realized by composing together software modules called *micro-protocols* into a software framework. This framework—essentially a runtime system—supports an event-driven execution model in which events are raised and fielded by the event handlers that make up the micro-protocols. This execution model is the key mechanism for supporting non-functional requirements since it allows, in essence, interleaved execution of modules. Furthermore, Cactus as well as other frameworks support dynamic adaptation of the micro-protocol configuration, that is, the runtime

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
AOSD 06, March 20–24, 2006, Bonn, Germany
Copyright 2006 ACM 1-59593-300-X/06/03 ...$5.00.

activation and deactivation of software modules [23, 38].

The parallels between the Cactus programming model and that used in AOSD and languages such as AspectJ are compelling—micro-protocols can be viewed as aspects and event handlers as advices, for example. While such analogies may be interesting in their own right, their true value lies in enabling lessons drawn from one domain to be applied in the other. The primary contribution of this paper is to do just that. This effort is especially timely since AOSD is currently being extended from its traditional programming language orientation to software systems such as operating systems, embedded systems, middleware, and distributed systems, areas that have long been addressed by researchers in CP. Furthermore, features such as runtime adaptation and customizable QoS attributes are gaining interest in the AOSD community.

This paper is organized as follows. Section 2 provides some background on AOSD, and reviews recent system-oriented developments in the area. Section 3 introduces Cactus and characterizes it in terms of AO computing. Section 4 outlines lessons we have learned—and that we think could be useful for future AO systems—in our work on using Cactus to implement modular configurable services ranging from low-level transport protocols to middleware and applications. Section 5 provides more extensive examples of the use of Cactus, and section 6 provides discussion on the relationship between AOSD and Cactus, on issues related to implementing customizable QoS attributes, and on related work on other CP frameworks. Finally, section 7 provides some concluding remarks.

2. BACKGROUND

Aspect Oriented Software Development (AOSD) [15] allows the separation of cross-cutting concerns into well-encapsulated entities called *aspects*. An aspect is a collection of *advices*, segments of code that collectively implement a particular concern (e.g., logging, security). In a non-AO system, this code would be scattered across numerous system modules, producing an entangled architecture that is difficult to develop, maintain, and extend. In an AO system, this code is modularized as aspects and systematically combined with the rest of the system through the process of *weaving*. Weaving is defined by the *joinpoint* and *composition models*. Joinpoints are points of interest in the execution of a program. The *joinpoint model* defines how joinpoints are structured and categorized, and how they relate to the execution of the system. In an object-oriented system for instance, the joinpoint model would typically include *object creations*, *method calls*, and *attribute access*. The composition model of an AO platform describes the mechanisms by which joinpoints and advices are brought together. This typically involves the use of a *pointcut description language*, i.e., a language that allows the definition of sets of joinpoints (called *pointcuts*) using quantifiers and predicates.

While originally limited to programming languages, AOSD has expanded to encompass system architectures, and is now being used as a structuring technique for system-level components such as middleware [30] and operating systems [2]. In such systems, the joinpoint model abstracts away from the actual implementation language being used and reflects higher-level entities of the system itself. For instance, in an operating system, the joinpoint model may cover scheduling events such as hardware interrupts and context switches, as well as process life-cycle events such as creation, blocking, and termination.

Composition models have also evolved to support these new uses of AOSD, covering issues such as how overlapping aspects are combined (ordering, compatibilities) and how aspects are instantiated (implicitly, explicitly). New models also address the interfacing of aspects with the base code, either directly through pointcut expression as in AspectJ [27] or indirectly through roles and weavelets as in CaesarJ [32].

Dynamic behavior has become increasingly important in modern AO platforms and has had an impact on both joinpoint and composition models. While early versions of AspectJ were only able to describe joinpoints as static source code locations [27], modern AO frameworks allow joinpoints that are defined by the (dynamic) execution of the underlying platform. For example, the *cflow* pointcut descriptor designates execution points that correspond to a particular call stack state (e.g., the creation of an object from class A only when invoked from a class B). The impact of dynamic concerns on composition models has also been significant. The instantiation model of aspects—when and how aspect instances are created—is now an integral part of AO frameworks. Dynamic weaving of aspects, where the activation or removal of an aspect is performed at run-time possibly as the result of the system's own computation process, is increasingly regarded as an elegant way to solve complex issues of dynamic adaptation and self-reconfiguration [7, 32, 35, 36, 34].

3. CACTUS DESIGN AND IMPLEMENTATION METHODOLOGY

Cactus is a design framework and runtime system for implementing and executing configurable services based on an event-driven execution model. Cactus has been used to build a wide variety of services ranging from low-level communication services analogous to UDP or TCP, to middleware services such as group membership and group remote procedure call, to application-level services such as a configurable distributed system monitoring service.

This section describes how Cactus is used to design and implement services where abstract service properties and QoS attributes are implemented as separate modules that can be configured together to provide customized service instances with the chosen set of properties.

3.1 Cactus design approach

The Cactus design approach for constructing highly-customizable services is illustrated in figure 1. First, the useful abstract properties or features of the service are identified. Useful features can be determined, for example, by studying existing implementations or designs for the particular service. Second, these features are implemented as configurable modules. The goal of the implementation is to maximize configurability, while minimizing the performance overhead, i.e., the performance difference between a monolithic implementation of a service and a configurable implementation that provides equivalent guarantees. Configurability is maximized by not introducing configuration constraints between the modules that implement abstract properties. Consider two abstract properties p_1 and p_2. If it makes sense for a service to have property p_1 and not p_2,

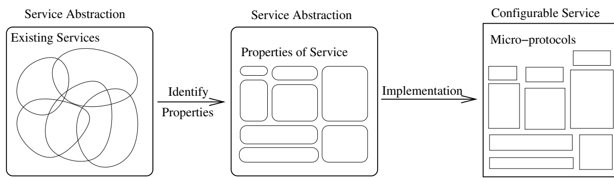

Figure 1: Design of a configurable service

p_2 and not p_1, and p_1 and p_2, then ideally the modules m_1 and m_2 implementing properties p_1 and p_2 should be able to be used separately or together. See [21] for details.

3.2 History of Cactus

Configurable communication protocols and other services such as file systems, database systems, and middleware have been around since System V STREAMS [40]. Such protocols are typically implemented as collections of modules that can be combined into different configurations, which allows their functionality to be customized based on application requirements and the characteristics of the underlying network. For example, one application may require that messages be delivered reliably and in order while another may not, or use of a given network might be optimized by using a particular type of congestion control. Most frameworks for configurable communication protocols dictate a layered, or hierarchical, composition model for the modules in which each module interacts only with modules immediately above and below it in the hierarchy. Examples of such frameworks include the *x*-kernel [26], Horus [39], and Ensemble [38].

While hierarchical protocol composition frameworks have been successful, we found them too limiting for the complex protocols often used to implement fault-tolerant distributed systems and services [22, 33]. Specifically, we found that modules that implement fine-grained logical properties of such protocols often need to interact more than allowed by strict hierarchical composition. In terms of AO programming, we observed that abstract service properties are often cross-cutting concerns that require processing at many different points of the message flow through a protocol or the flow of a service request through a service. Therefore, we developed a more flexible composition model based on event-driven execution. This model, implemented in the Cactus system (as well as its predecessor, Coyote [4]) allows "interleaving" of module execution by using events as the primary control flow mechanism.

3.3 Cactus programming model

The Cactus model is based on a two-level view of system composition as illustrated in figure 2: a system is constructed from services, where each service is then composed of modules that implement the abstract properties of the service. Services are composed using the traditional layered approach, while the modules within services—called *micro-protocols* in Cactus—are organized non-hierarchically and interact with one another by raising and fielding events. In contrast with the traditional layered composition approaches, the modules do not need to be linearly ordered and their interactions can be much richer. In fact, this model allows the execution of different modules to be arbitrarily interleaved. The result is a model that supports flexible interaction and data sharing between modules, but

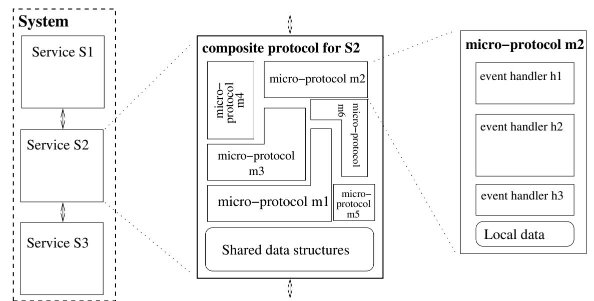

Figure 2: Two-level composition model

also allows the strict separation and proscribed interaction through a uniform protocol interface between independent services where appropriate.

Figure 3 illustrates in more detail a configurable service implemented using Cactus, in particular, a Configurable Transport Protocol (CTP) [46]. A service is implemented as a *composite protocol*, with each service property or other functional component implemented as a micro-protocol. A micro-protocol is, in turn, structured as a collection of *event handlers* that are executed when a specified event occurs. Events are used to signify state changes of interest, such as "message arrival from the network". When such an event occurs, all event handlers bound to that event are executed. Thus, Cactus imposes an event-driven design and programming paradigm, where a micro-protocol designer has to decide what events are relevant for the micro-protocol being implemented and for each such event, what are the actions that this micro-protocol should take when the event occurs.

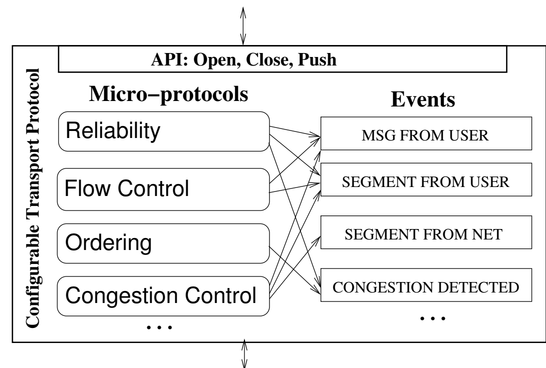

Figure 3: CTP composite protocol

While the event-driven composition approach could be implemented as a new event-based programming language or as event-based extensions to an existing programming language, we have chosen to implement the model as a set of conventions and a software library. This has made it possible to implement Cactus in different programming languages including C, C++, and Java. The Cactus library provides a variety of operations for managing events and event handlers, including operations for binding a handler to a specified event and for raising an event, which causes all the handlers bound to that event to be executed. An event can also be raised with a specified delay to implement time-driven execution (delayed event), and with either blocking or non-blocking semantics on the thread raising the event.

The order of handler execution can also be specified if desired. Other operations are available for unbinding handlers, creating and deleting events, halting event execution, and canceling a delayed event. Handler execution is atomic with respect to concurrency, i.e., a handler is executed to completion before any other handler is started unless it voluntarily yields the CPU.

Micro-protocols are created dynamically, typically at service startup time but also during execution to implement adaptive behavior [23]. Furthermore, multiple instances of the same micro-protocol can be created and exist simultaneously. Using multiple instances of the same micro-protocol is useful in cases such as group communication—one instance of a micro-protocol is created for each communication partner—or cryptography—multiple instances of the same DES micro-protocols can be instantiated to create semantics similar to 3DES. Micro-protocols can have arguments and often maintain some internal state.

Event handlers have two types of arguments, *static arguments* passed to the handler in the bind operation and *dynamic arguments* passed to the handler in the raise operation. Thus, the static arguments remain the same for each execution of the handler, while the dynamic arguments change for each execution. Static arguments are typically used to give direction to the micro-protocols execution (e.g., an encryption key to be used by the micro-protocol), while dynamic arguments are typically messages or service requests received by the service.

Events can be passed as arguments to micro-protocols at initialization and as static or dynamic arguments to event handlers.

3.4 Example

Figure 4 provides a simple example that illustrates some of the features of Cactus. This figure shows a very simple micro-protocol (SampleLogger) for message logging consisting of one event handler (LogMessage) and an initialization section. This micro-protocol can be used to log a sample of messages originating from a set of specified IP addresses (`sources`), specifically, one in every `freq` messages. The event handler illustrates the use of both static and dynamic arguments; the former include the IP address to monitor passed in the bind operation, while the latter includes each message that arrives. Specifically, since the handler LogMessage only checks messages from one IP address, the micro-protocol binds the same handler with different IP addresses for each IP address specified in the set. Thus, when the event occurs, each bound instance of the handler checks the message for the source IP address specified for this handler. Naturally, this is a simplified example, since such an implementation could be very inefficient in practice if the sets of IP addresses are large.

The example also illustrates that the event to which the handler is bound can be passed as an argument to the micro-protocol. This allows the same micro-protocol to be used for different events indicating message arrival. Finally, the figure shows how a group communication protocol could create several instances of this micro-protocol to log messages from different group members at different frequencies.

Note that the example micro-protocol code in figure 4 uses a pseudo-code notation for composite protocols, micro-protocols, event handlers, and event handling operations. As indicated above, the Cactus model is implemented in each programming language as a library of event handling operations and a runtime system, as well as a set of conventions on how the logical concepts in the model are mapped to concrete entities provided by the underlying programming language. Thus, the concrete syntax of event handling operations and parameter passing are different depending on the language.

```
micro-protocol SampleLogger(int freq, IPset sources,
    event Ev) {

  int counter, myFreq;

  handler LogMessage(IP sid, Message msg) {
    if (msg.source == sid) {
      counter++;
      if (counter == myFreq) {
        System.log(toString(msg)); counter = 0; }
    }
  }
  initial {
    counter = 0; myFreq = freq;
    for each source in sources do {
      Ev.bind(LogMessage, source); }
  }
}
protocol GroupCommunication(int groupID) {
  initial {
    msgFromGroup = new Event("MsgArrivalEvent");
    dangerSet = new IPset("123.456.78.90", ...);
    friendlySet = new IPset("321.654.87.09", ...);
    new SampleLogger(10,dangerSet,msgFromGroup);
    new SampleLogger(100,friendlySet,msgFromGroup); }
}
```

Figure 4: Example micro-protocol

3.5 Mapping Cactus to AOSD concepts

The different elements comprising the Cactus programming model can easily be viewed from an AO perspective. Micro-protocols are encapsulating units that contain event handlers, and can be seen as aspects. Event handlers consume event occurrences, and can been seen as advices. Event occurrences that result from a raise operation represent a particular point of interest in the execution flow of the system, and can be seen as joinpoints, and event raise sites as joinpoint shadows. By binding to an event type (simply referred to as "an event" in the following) a handler receives all occurrences of this event, making events comparable to pointcuts and the event binding mechanism comparable to a composition model. In this respect, Cactus is very similar to the concept of event-based AOP that has recently emerged [14].

Unlike AOSD, however, Cactus does not provide a pointcut description language or a quantification mechanism other than the use of event names. While a handler can in principle be bound to several events to give a disjunctive combination of pointcuts—the handler is executed if event A *or* event B occurs—a given handler is more commonly bound only to one event, with separate handlers within a micro-protocol used to handle different events. The overall functionality of the micro-protocol thus truly cross-cuts the implementation of the service or protocol.

4. LESSONS FROM CACTUS

In this section, we describe a set of features we have found useful in Cactus when constructing distributed customizable

services and that we believe could be instrumental when applying AO programming in such domains. For each feature, we discuss the feature and its use in Cactus, and compare it to the mechanisms available in current AOP platforms. These features also form the backdrop for sections 5 and 6. In section 5, we present two concrete services that have been implemented using Cactus, while in section 6 we discuss the respective merits of Cactus, AO, and other approaches to realizing configurable distributed services.

4.1 Application defined joinpoints

While AOP traditionally defines joinpoints syntactically in terms of programming language entities such as variables, procedures, and classes, we have found that joinpoints or events that convey more application-specific meaning are often required. For example, consider a configurable reliable multicast service with micro-protocols ensuring attributes such as reliability, FIFO ordering, and consistent total ordering of messages. In such a service, a stability micro-protocol is often used to keep track of when each message has been received by all group members, that is, when the message becomes *stable*. Numerous other micro-protocols use message stability to trigger actions; for example, a garbage collection micro-protocol can safely delete the message once it is stable or an archiving micro-protocol might move stable messages to disk. Having the stability micro-protocol raise an event when a message becomes stable is an easy way of communicating this information to an arbitrary number of other micro-protocols.

AO systems do not directly support such "semantic" joinpoints, although there are several possible ways to achieve a similar effect. One is to trigger a dynamic joinpoint in the base program by creating an object or updating a field, with the convention that this joinpoint should be interpreted with a particular semantics (e.g., a message is now stable). A variant of this approach is to use a naming convention that carries application-specific semantics. For instance, a pointcut expression like *call(* *.print*(..))* can match any method printing information on a console, *provided* that the original program followed the convention that any such method and only these methods start with "print".

Embedding higher-level semantics in names can, however, become laborious when multiple facets must be encoded or when additional organizational naming conventions must be taken into account. One solution can be to structure the program so that names only need to expose one piece of semantic information at a time. This can, however, lead to a counter-intuitive structure and a general fragmentation of the whole program. This name-oriented approach also makes it difficult to modify the semantics attached to a joinpoint. For example, suppose that, in addition to the "*print*" convention, methods that handle confidential information must contain the string "*Confidential*". If the program is changed so that a print method processes confidential information where it previously did not, all locations calling this method must be changed along with the method's name.

Overall, we view such workarounds as awkward ways of achieving the desired effect and believe that allowing the specification of such semantic joinpoints would be useful for AOSD. The general interest raised by annotation-based pointcuts, in particular with the new Java meta-data facility provided by Java 1.5, is a step in this direction [10, 43, 34, 6]. Such annotations can convey semantic information in much the same way as Cactus events can.

4.2 Asynchronous joinpoints

Cactus supports two modes of event raise, *synchronous* and *asynchronous*. In the synchronous case, the handlers bound to the event are executed before control returns to the micro-protocol that invoked *raise*. This is similar to typical AO approaches where all advices are executed before the base functionality continues execution.

In contrast, in the asynchronous case, control returns to the invoker of the raise immediately, possibly before the handlers are executed. Asynchronous event occurrences have turned out to be useful both as a programming tool and when optimizing the execution of a composite Cactus protocol. As a programming tool, we have found asynchronous events to be very useful in protocol engineering, and we anticipate they would also be useful in other contexts such as logging or profiling where data collection does not need to be tightly synchronized with execution of the base program.

Asynchronous event execution also makes it easier for the runtime system to optimize execution since handlers do not need to be executed immediately. Our different Cactus implementations (in C, Java, and C++) have exploited this factor by using different strategies for scheduling asynchronous event execution. One implementation strategy is simply to rely on the multi-threading capabilities of the underlying operating system or JVM. It is also possible to implement lightweight schedulers inside the Cactus runtime system to optimize predictability and performance. In fact, the actual implementation has no impact on the programming model as long as it respects the Cactus event-handling semantics. This is particularly useful when there is limited threading support provided, as might be the case in embedded systems or operating system kernels. For example, we used deadline-based non-preemptive scheduling implemented by an event queue and a single dispatcher thread in our real-time version of Cactus [24].

To our knowledge, only the CASS platform provides asynchronous joinpoints comparable to Cactus asynchronous events [11]. While the CAM/DAOP platform allows advices activated at the same joinpoint to execute in parallel, control only returns to the base program when the last advice terminates [35]. It is interesting to note that both CASS and CAM/DAOP have been developed in the context of distributed aspects, in which parallel and asynchronous execution are natural choices. Traditional AO systems could emulate asynchronous joinpoints by explicitly creating a new thread that then executes the real function of the advice. This is not very elegant, however, and it also leaves all scheduling decisions to the underlying OS or JVM scheduler, thus precluding any domain-specific strategies as done in Cactus.

4.3 Time-driven aspects

The passage of time is an important concern for many micro-protocols implementing QoS attributes. For example, a reliability micro-protocol needs to retransmit a message after a time period or a checkpointing micro-protocol needs to store the program state at some time interval. In Cactus, time-driven execution is implemented by allowing asynchronous events to be raised with a specified delay, that is,

rid = event.raise(. . ., delay);

This "delayed raise" returns a handle to a (future) occur-

rence of the event that can be used to cancel this occurrence if necessary. This is particularly useful for implementing watchdogs. For example, a timed event can be raised when a message is sent that requires an acknowledgment, and then canceled when the acknowledgment arrives. As a result, if no acknowledgment arrives in time, the timed event is executed and some remedial action can be taken.

To the best of our knowledge, no AOP platform explicitly supports time-triggered joinpoints. This obviously cannot be realized by delaying an advice with a sleep function, since the advice would then block the base program and any other pending aspects. A possible workaround is to spawn a new thread from within the advice, but this is far less satisfactory than the direct programmatic support of time constraints.

Time-related concerns could be added in AOP platforms using a number of approaches. One is to extend asynchronous joinpoints with a specified delay. Another possibility is to extend AOP platforms that support state-machine based pointcuts (e.g., JAsCO [43]) or event-based AO frameworks (e.g., [14]) with temporal pointcut descriptors to specify temporal properties. However, these approaches alone would not permit the cancellation of an asynchronous joinpoint analogous to event cancellation in Cactus, something that would probably require the ability to handle joinpoints as first-class entities. We return to this last point below in section 4.8.

4.4 Parameterized aspects

We have found it very useful to be able to specify arguments to event handlers both at binding time ("weaving") and when event occurrences are raised ("joinpoints"). In Cactus, these correspond to the *bind* and *raise* operations. When an event is raised, the event handler gets two sets of arguments: static arguments assigned at bind time and dynamic arguments assigned at the time the raise operation is executed.

The passing of arguments at raise time in Cactus corresponds to the extraction of context information from joinpoints (e.g., arguments, target, this) through pointcut primitives in AOP frameworks. However, even in this case, there is usually no way to specify that some particular information not available in the joinpoint context should be directly passed to an advice. If an aspect needs additional information to be collected, this can be solved by inserting additional attributes to the impacted objects (assuming an OO environment), and using context information to access it.

The passing of bind arguments in Cactus corresponds to parameterization of aspects at weaving/deployment time. This feature is usually only supported by AO systems that make a clear distinction between aspects and their binding to a base program. AO systems that support aspect parameterization typically either support explicit aspect instantiation such as CaesarJ [32], the passing of arguments to pointcuts such as JAC [34] and AspectJ2EE [9], or have template-like facilities such as the configuration scripts in JBoss [7].

4.5 Dynamic reconfiguration

The ability to bind and unbind handlers at runtime has proven to be useful in many micro-protocols. Activating and deactivating micro-protocols in this way is a natural mechanism for implementing adaptive services, that is, services that change their behavior at runtime based on changes in the application requirements or the runtime environment [23]. Such a mechanism is also useful for many non-adaptive services, since many protocols and services have distinct execution phases (initialization, normal execution, failure handling, shutdown) where different behaviors implemented by different handlers may be useful.

Dynamic binding and unbinding of handlers is similar to the dynamic deployment and removal of aspects [35, 7, 32, 36, 34]. There is, however, a difference between (i) approaches that use dynamic weaving and can dynamically deploy aspects using pointcuts that were unplanned when the base program was compiled (e.g., [34, 7]), and (ii) approaches that statically insert hooks for pointcuts known in advance, but allow for the corresponding aspect to be dynamically deployed on these pointcuts (e.g., [32]). The Cactus model is very close to the latter since the joinpoints are defined by the points at which events are raised in the code, while the set of handlers (advices) to be executed at each event occurrence can change at runtime when micro-protocols bind and unbind their handlers. Additionally, however, Cactus allows new micro-protocol code to be loaded into a running protocol (e.g., using dynamic libraries in C or dynamic class loaders in Java), and these new micro-protocols may create new events and raise these new events at runtime.

A complication resulting from such flexibility is that event handling semantics must be well-defined when handlers are being bound and unbound at the same time events are being raised. Timed events in particular can be raised long before the corresponding handlers are executed, increasing the time span any synchronization must cover. While the Cactus execution model ensures that the execution of each event handler individually is atomic, the collective execution of all the handlers bound to an event is not. Therefore, it is possible that a handler can be unbound after the event is raised, but before this event handler has been executed. To address this issue, Cactus determines the set of handlers to be executed when the event is raised and executes this set even if the bindings have subsequently changed. This approach minimizes interference between concurrent micro-protocols and has proven to be a sound basis for supporting composition.

While the ability to activate and deactivate micro-protocols at runtime is necessary for building adaptive services, it is often not sufficient. For example, coordination is often required (i) locally between composite protocols across different system layers (*inter-layer coordination*), and (ii) globally between peer composite protocols located on different hosts (*inter-host coordination*) [8]. Inter-layer coordination is required to ensure that different adaptive protocols in the same protocol stack do not perform conflicting adaptations or overadapt by each reacting to the same change. An example where such coordination is needed is when video is transmitted over a wireless network and the underlying transport protocol can adapt to network losses by introducing redundancy in the transmission in the form of forward error correction. As a result, the video application has to adapt the amount of video data it sends per time unit, since less bandwidth is available for the video data.

Inter-host coordination, on the other hand, is required to ensure that peer protocols can understand one another's messages. For example, if one peer adapts to a suspected security threat by starting to encrypt messages, it is ob-

vious that its peers must also adapt so they can decrypt the message body using the correct key. We have explored distributed adaptation coordination algorithms that allow underlying composite protocols to adapt transparently to higher-level services and applications [8].

4.6 Ordering of advices

When more than one handler is bound to the same event, the order in which they are executed may be important. This became obvious in an early implementation of the event-driven execution model in which handlers were simply executed in the order they happened to bind to the event. The result was that programmers ended up manipulating the order in which micro-protocols were initialized in order to provide some sort of explicit ordering between handlers. In addition to being ad hoc, this approach lacks flexibility since it enforces the same order for all events.

In Cactus, explicit handler ordering is enabled by allowing the programmer to specify an `order` argument in the bind operation:

event.bind(handler,static arguments, order);

This argument is simply an integer value that is used by the runtime system to order handler execution in the obvious way. While sufficient, this approach has proven to be somewhat difficult to use in practice since it requires programmers to keep track of which integers have been used by which micro-protocols for each event. We are exploring better options, including specifying just the required ordering constraints (e.g., "handler h_1 of micro-protocol m_1 has to be executed before handler h_2 of micro-protocol m_2 for event e_1") and then having the system generate an order that satisfies these constraints for all the micro-protocols that are included in a given service configuration.

A number of solutions for aspect ordering has been proposed, and the different approaches explored in Cactus have their counterparts in AOSD frameworks. For example, ordering based on aspect instantiation is used in CaesarJ [32]. Somewhat more flexible are priorities in PROSE, which decouple the initialization order from the aspect order [36]. A mechanism similar to Cactus is supported only by a few frameworks, since it requires the ability to specify orders between advices at the pointcut level and an explicit notion of binding. For example, JBoss permits advice ordering per pointcut with the `<stack>` construct [7]. JAsCO has a similar functionality based on *hooks* and *connectors* [43].

4.7 Relations between aspects

While the work on AOP has traditionally focused on individual aspects, the configurable services implemented using Cactus have large numbers of micro-protocols that are designed to operate together in different combinations. It is obvious that all combinations are not possible. For example, the fundamental properties provided by different micro-protocols might be in conflict (e.g., FIFO order vs. immediate delivery of "important" messages). In other cases, the combination might be logically appropriate, but the implementations of the micro-protocols might be incompatible. To address these issues, we have defined a set of relations between micro-protocols (independence, dependence, conflict) that describe the compatibility between micro-protocols. We have also developed a configuration support tool that enforces these constraints [21].

Many AOP platforms provide some more or less direct means of enforcing dependencies and incompatibilities between aspects. JAC for instance provides a composition aspect that can be configured to encapsulate such relationships. To our knowledge, however, only JAsCO and EAOP directly support the ability to specify inter-aspects relationships in a *declarative* manner. For example, EAOP allows one to specify that an aspect X should only be applied at a joinpoint if another aspect Y has been applied at the same joinpoint [30]. Only JAsCO offers an *exclude* operator.

As aspects grow in popularity, the importance of reusable aspect libraries is expected to grow and we expect the ability to declare relations such as exclusion or dependency between aspects will become increasingly important.

4.8 Joinpoints as first class objects

As discussed above in the context of timed events (section 4.3), the ability to manipulate events as first class entities allows the implementation of complex time-driven control flows in Cactus. Unlike events, traditional joinpoints are not first-class entities and thus do not support this kind of treatment. They cannot be created, deleted, stored in data structures, or passed as arguments. AO frameworks such as AspectJ, CaesarJ, PROSE, and JAsCO [34, 32, 43, 36, 27]) have special reflective elements such as *thisJoinpoint* in AspectJ or *Invocation* in JBoss that represent joinpoints at runtime. However, these mechanisms only provide observation capacities. Joinpoints cannot be instantiated, and they usually cannot be forwarded or morphed into a different joinpoint type.

Treating joinpoints as first class entities (i.e., that can be "thrown" or "raised" explicitly) would actually reflect some of the strategies used to implement aspects. In Prose [36], for instance, a lower layer observes the base program and generates joinpoint occurrences accordingly. These joinpoint instances are then consumed by a higher level to activate the appropriate advices, in a way similar to event handlers in Cactus.

4.9 Instance specific aspects

In communication protocols, a distinction is often made between a *protocol*, which provides the procedures and data structures for interaction, and a *session*, which is an instantiation of the protocol used to create a logical connection between communicating peer protocols. From an object-oriented point of view, the protocol can be viewed as a class while the session is an instance of this class. Often, multiple sessions are created using the same protocol, for example, to form connections to different communication partners or connections for different purposes between the same parties (e.g., a control channel and a data channel for transmitting streaming video). In many cases, the requirements for the different sessions are also often different.

Cactus allows different sessions of the same configurable protocol to have different properties by activating different sets of micro-protocols for each session. For example, a video application that uses our Configurable Transport Protocol (CTP) (see figure 3) could open two separate sessions, one for use as a control control and the other for use as a data channel. The control channel might have reliability and FIFO ordering micro-protocols, but without congestion, flow, or jitter control. The data channel, on the other hand, might have micro-protocols for rate-based congestion con-

trol and encryption-based privacy, but without reliability and ordering [46].

In AO terms, this ability to have sessions with different properties would roughly correspond to instance-local aspects such as introduced in the Steamloom system [5].

5. CACTUS EXAMPLES

Here, we describe two concrete Cactus services as a way to illustrate the lessons from the previous section. The first is a service that allows QoS attributes related to fault tolerance, timeliness, and security to be customized for distributed object platforms, while the second is a communication channel abstraction that provides dependability and real-time guarantees. A similar discussion could be derived from virtually any of the many Cactus services that have been implemented.

5.1 QoS for CORBA and Java RMI

Cactus has been used to enhance existing application-level CORBA and Java RMI services by imposing a QoS service, called CQoS, between the client and the middleware (CORBA ORB or Java RMI), and between the middleware and the application-level server implementation [18]. CQoS is transparent to both the client and server and no modifications are required in either the client or server code, or in the IDL description of the server interface. This transparency is achieved by replacing the original stubs generated by the CORBA and Java RMI tools with custom generated stubs that include Cactus composite protocols (called CactusClient and CactusServer, respectively) that are invoked when the requests and replies pass through the CQoS stubs (see figure 5). Note that the same CactusClient and CactusServer composite protocols work for both CORBA and Java RMI, with just the stubs being different for these different middleware platforms.

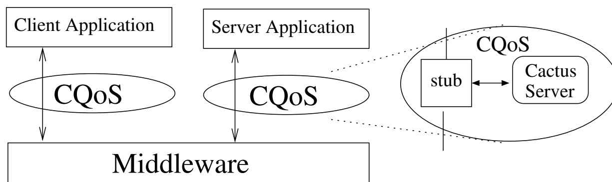

Figure 5: Customizing QoS with CQoS.

The set of events used in this Cactus service reflects the flow of requests and replies through the client and server side composite protocols. For example, when the client issues a new request, the client-side stub invokes CactusClient, which raises the newRequest event. Similarly, on the server side, the CactusServer raises the newServerRequest event when the server-side stub notifies it of the new request. Other events on the client side include readyToSend, which indicates that a request is ready to be sent to server(s), and invokeSuccess and invokeFailure, which indicate that an invocation completed successfully or failed, respectively. Other events on the server side include readyToInvoke and invokeReturn, which indicate that an invocation is ready to be passed to the servant and that an invocation has completed, respectively, and RequestReturned, which indicates that a reply to the client request has been sent to the client.

The micro-protocols in this service include `ClientBase` and `ServerBase` that implement the base functionality of passing the requests and replies through the composite protocols at the client and server sides, respectively. Each consists of three handlers. Other optional micro-protocols provide enhancements to the service QoS. For fault tolerance properties, `ActiveRep` and `PassiveRep` micro-protocols implement active and passive (primary-backup) type replication of the request to multiple servers to mask server crash failures, `MajorityVote` implements voting to deal with server value failures, and `TotalOrder` ensures requests from multiple concurrent clients are processed at the same order at different replicas. For security properties, the services includes micro-protocols for privacy (DES encryption of request and replies), integrity (MD5 checksums), and access control based on access control lists. Finally, for timeliness properties, the service includes a scheduling micro-protocol that maintains separate request queues for low and high priority requests and for throttling the number of concurrently active low priority requests. Note that while such a timeliness micro-protocol cannot ensure that (soft or hard) deadlines can be met, it can provide service differentiation between low and high priority requests.

This service uses many of the features outlined in the previous section. First, ordering of handlers for each event is critical to weave the micro-protocol functionality properly at each event. For example, it can be important that some handler is executed as the last one or the first one for a particular event. Static arguments are used extensively by the `ActiveRep` micro-protocol by binding a given handler once for each server replica with the server id as the static argument. Although this protocol does not use delayed events, it uses asynchronous (non-blocking) raise for the RequestReturned event.

5.2 Real-Time Dependable (RTD) channels

Cactus has also been used to implement services with (hard) real-time attributes. These implementations naturally rely on the underlying operating system supporting real-time scheduling and resource allocation, in our case the Open Group/RI MK 7.3 Mach real-time operating system. The RTD (Real-Time Dependable) Channel is an example of such a service [24, 12]. An RTD Channel can be customized along a number of dimensions, including the communication topology (ranging from point-to-point unidirectional communication to many-to-many group communication), the reliability of message transmission, message ordering properties, and the real-time deadlines associated with message transmission along the channel.

While reliability and ordering properties are implemented as micro-protocols as in any Cactus protocol, the real-time deadline guarantees have to be implemented by using admission control, resource allocation, and real-time scheduling provided by the underlying operating system. The admission control decides if the request for creation of a new RTD channel can be accepted given the current set of existing channels in the distributed system and the specific requirements of the new channel. The resource allocation component in this case determines the priority for the channel, which is then used by the OS to schedule the processing of messages in different channels in the system. The ability to customize a variety of properties in creating an RTD channel complicates realization of the real-time aspects of the service. For example, to implement message deadlines it

must be possible to predict the CPU time required to process each message, as well as other resource requirements such as those related to network bandwidth and memory usage. The choice of properties affects all these calculations since the choice of micro-protocols impacts the number of messages sent and processed by the channel (e.g., acknowledgment messages), which again affects timeliness. The details of the admission control and resource allocation can be found in [12].

The set of events used by the composite protocol consists mostly of events related to message arrival from (and delivery to) the application and the network. However, this protocol uses separate events for the arrival of real-time data messages, non real-time data messages, and control messages. These specialized events are raised by the base micro-protocol based on the message header fields, and they allow the other micro-protocols to be notified only when a relevant message has arrived.

The RTD Channel composite protocol includes a base micro-protocol, two alternative micro-protocols for message reliability, three for message ordering, and one to force out of order message delivery. For reliability, the micro-protocols are `MulReliable` that sends each message multiple times over the network and `AckReliable` that uses positive acknowledgments, i.e., transmits a message repeatedly until an acknowledgment is received from the destination. For ordering, the micro-protocols are `FifoOrder` that uses sender assigned sequence numbers to deliver messages in the same order they were sent, `CausalOrder` that preserves causality in message delivery,[1] and `TotalOrder` that ensures all messages are received in the same order on all destinations. Finally, the `ForceUp` micro-protocol forces the delivery of a message out of order if its deadline would otherwise be missed because earlier messages in the specified order are missing (and thus, preventing its in-order delivery).

This protocol uses a number of the features discussed in section 5. Specifically, is relies on handler ordering extensively, it defines semantic joinpoints (the message specific events), uses timed events (for `AckReliable` and `ForceUp`), and it relies on session specific configurations of micro-protocols (each channel instance is a separate session of the same RTD Channel protocol). Finally, some of the micro-protocols have relations that dictate their configuration constraints; for example, certain ordering protocols can only be used with specific channel topologies.

6. DISCUSSION

6.1 Comparison

Based on the two Cactus examples and the features presented in section 4, we discuss some of the more fundamental differences between Cactus and AO environments. One such difference is that aspects tend to be much more generic, that is, they can be applied to different systems or classes, while micro-protocols are typically specific to the composite protocol implementing a service. Some of the reasons for the specificity are "syntactic" in the sense that micro-protocols are designed to operate with a certain set of events (event names), and assume certain shared data structures and type definitions for the messages processed by the service. Other reasons are more fundamental in nature. For example, the way in which reliability is implemented differs depending on whether the service provides unidirectional message streams (e.g., streaming video) or a request-reply paradigm (e.g., remote procedure call). The AO approach, where pointcuts are specified separately from the aspects either using abstract pointcuts or through an explicit binding mechanism, could be applied to Cactus to make micro-protocols more portable by eliminating the need to encode specific event names. As illustrated by the example in figure 4, this can already be partly achieved by passing event names as arguments to micro-protocols.

Another major difference is naturally that Cactus requires *raise* operations (joinpoint shadows) and event types (pointcuts) to be programmed explicitly into the code they crosscut, whereas almost all AOP languages provide quantification mechanisms (pointcut description languages) to specify joinpoint sets with no modification of the base program. The trend towards annotation-based aspects (section 4.1) suggests that the type of functionality provided by the Cactus approach is also useful in an AOP context. The value is especially clear in cases where "implicit" pointcuts (i.e., limited to the properties of programmatic entities) are not desirable or even possible.

Conversely, implicit event reification could be useful in Cactus. For example, a non-trivial number of events in a typical composite protocol are associated with messages or service requests arriving and leaving the protocol. Such standardized "events" could be reified transparently in the same way as AO joinpoints. Adopting AO techniques to extract events from an existing system would also remove the need to attach Cactus manually to the stubs or proxies that wrap a service, thus easing the direct application of Cactus to existing programs.

At the level of the composition mechanisms, Cactus is closer to multi-dimensional separation of concerns [45] than to traditional aspect orientation. Cactus does not explicitly distinguish between a "base program" and cross-cutting entities (aspects). Instead, Cactus services are constructed purely out of micro-protocols. However, in practice, Cactus protocols tend to be structured around a "base" micro-protocol that implements the core service functionality and is required in each configuration of the system. Cactus has also been used to enhance existing services by attaching composite protocols to an existing service stack [18, 19]. Both strategies, although unrelated to the Cactus composition model, are strongly reminiscent of the base/meta separation found in standard AO systems.

Finally, note that, although the event-based weaving of Cactus differs from most AO platforms, the underlying mechanisms are in fact very similar. For instance, we have done work on weaving the handlers "permanently" by inlining the handler code at the event raise sites when dynamic profiling shows that the set of handlers for an event raise site remains invariant [37]. If the handler binding for the optimized raise site changes, the optimized code falls back to the normal event handling procedure. Such optimization improves the performance of a customizable service, but for only one configuration of the service at a time.

6.2 Issues in QoS composability

While encapsulating simple concerns such as logging or

---

[1] If the sending of message m_2 was potentially caused by the reception of message m_1, then m_1 is delivered before m_2 at all destinations.

debugging into reusable aspects or micro-protocols is easy, factoring out "systemic" QoS attributes such as reliability, security, or timeliness by the same means can be much more challenging. Hard real time is perhaps the best example of a guarantee that cannot simply be provided by a single "real-time" micro-protocol. Limited enhancement of some soft timeliness properties is possible by having micro-protocols manipulate thread priorities and order messages or requests based on deadline information. Rigorous hard real-time guarantees cannot, however, be achieved without taking into consideration the complete protocol stack and indeed, the entire system. This is required to derive predictable execution times and manage resource allocation appropriately. The use of a configurable micro-protocol framework, where each micro-protocol might or might be not activated, makes this especially difficult. In particular, each configuration of the service typically has different CPU requirements, different resource needs, and different worst case execution times.

We have addressed some of these issues in a real-time version of Cactus [24] and in the RTD Channel service described in section 5.2. We had to solve two main problems. First, for communication services, the service configuration not only dictates the processing time per message, but also has an impact on the number of messages sent and received. For example, a reliability property may require message retransmissions. Second, the problem is further complicated by the fact that the number of messages often depends on the number of communication participants. We addressed these issues by adding a service-specific Admission Control module that encapsulates all required resource calculations and then makes a resource reservation for the specific service configuration. This allows the configurability of a service implemented with the micro-protocol framework to be separated from global system resource allocation issues.

Security properties such as communication security and access control are similar to real-time properties to the extent that simply adding an encryption (or access control) micro-protocol does not necessarily make the whole system secure. Such micro-protocols can only strengthen a part of the system and may leave other vulnerabilities open that allow a security breach. Implementing security mechanisms as separate micro-protocols is, however, far easier than hard real-time properties. For example, we have built a configurable secure communication service called SecComm, in which each security property, including privacy, integrity, authenticity, non-repudiation, and replay prevention, can be provided by a choice of micro-protocols or combinations of micro-protocols [25]. The challenges include the fact that different security transformations have a number of dependencies and ordering constraints, and that the level of security achievable by using multiple cryptographic methods is still an unsolved problem.

In contrast to real time and security, our experience has been that fault-tolerance properties are relatively easy to realize as independent micro-protocols. For example, replication or retransmission micro-protocols can be implemented to add tolerance to host and communication failures. Often, it is necessary to hide the effects of such techniques, namely multiple replies and duplicate messages, from the application and maybe even other micro-protocols. This is easy to do by adding a voting or duplicate elimination micro-protocol.

Finally, we do not see any fundamental impossibility in providing combinations of fault-tolerance, security, and timeliness attributes. As illustrated in this paper, some of the Cactus services in fact provide various subsets of these attributes. The attributes naturally have an impact on one another, as illustrated by how the reliability properties impact timeliness in the RTD Channel service. Similarly, security attributes implemented using cryptography have an impact on CPU utilization (per message) and thus, the deadline guarantees. It is often argued that the redundancy used to provide fault tolerance makes the system less secure by introducing more vulnerable points into the system (e.g., copies of the same file on multiple computers), but prior work has shown that this does not need to be the case if security and redundancy are designed jointly [13].

6.3 Related work on configurable protocols

In this section, we describe other approaches to constructing configurable system-level software (e.g., protocol stacks, databases), and, where possible, contrast them with AO concepts.

Layered approaches. These approaches are oriented towards constructing systems as a stack or directed graph of modules, where each module typically interacts only with modules immediately above and below it in the hierarchy. Examples of this approach include the *x*-kernel [26], Horus [39], the Genesis database system [3], and stackable file systems [20]. Some of the approaches require that all modules export an identical interface, e.g., *x*-kernel, while others allow layer-specific interfaces, e.g., Genesis. Layered approaches result from the application of modularization to the problem of configurability with the goal of organizing the different building blocks in a manner that is at the same time flexible (to ease configurability) and principled (to guarantee good software quality). Hierarchical approaches are not in this respect particularly related to aspect orientation. Rather, they focus on separation of concerns in that they encapsulate different features into different modules, and allow a composite service with configurable properties to be built as a combination of these modules. Possible cross-cutting and interference issues are, however, not dealt with in any generic manner.

Slotted approaches. In these approaches, a customizable software component is constructed as a fixed system backplane with slots that can be filled using a choice of modules for each slot. An example of this approach is Adaptive [41]. Slotted approaches can be seen as a forerunner of *object-oriented frameworks* as defined in [16]. In contrast with a hierarchical approach, modules in this approach are typically typed and thus, can only be used in one specific slot. Similarly, each slot in the system must be filled with a given type of module, which means that the backplane and each of the sets of modules depend on one another. In some systems such as Adaptive, a slot can be filled with a *composite* module, which is essentially a smaller backplane with new slots. ¿From an AO point of view, each slot can be seen as a pointcut and each module as an aspect. The backplane has the same role as the base program in an AO system, providing the structure into which modules are inserted. Slotted approaches are, of course, far more limited that AO systems, since slots are decided at design time and must be explicitly built into the system, unlike pointcuts. The slot-backplane binding also does not support any notion of quantification. A slot can only have one module associated with it, un-

like joinpoints that can be targeted by several aspects. The composite modules alleviate this limitation to some extent by allowing a level of recursion, i.e., in AO terms, by allowing one or more aspects to be attached to an aspect rather than to a base program.

Class hierarchy based approaches. In these approaches, the mechanisms for constructing a customized software component are presented to users as an object class hierarchy. A predefined class hierarchy specifies the available components, which can then be manipulated by invoking the object methods. New classes can be defined as derived classes of existing ones. Examples of this approach are Arjuna [42] and the configurable mixed-media file system described in [31]. While these approaches do not support joinpoints or weaving, the modules take advantage of inheritance and subtyping similar to object-oriented AOP languages.

Interception- and reflection-based approaches. These approaches typically rely on a distributed object model in which distributed objects communicate with each other by message passing and in which interception is used to configure message communication paths transparently to the application [1]. Interception-based systems have given rise to full-fledged reflective infrastructures that have been used to add non-functional features such as fault-tolerance (replication) and security (encryption, authentication) transparently to existing distributed applications [17, 1, 29]. Message processing events (sending, reception) here are comparable to the joinpoints of an AOP framework. There is, however, no explicit notion of pointcuts, and if needed, quantification must be implemented using reflective capabilities to decide, for instance, whether a message needs to be intercepted. These approaches go much further that the previous ones by allowing a true cross-cutting (1-n) composition between non-functional mechanisms and the base system.

Reflective approaches have been further refined to address middleware architectures. For example, multi-layer reflection has been proposed to overcome the complexity of modern distributed platforms [44] (i.e., the *thickness* of software). Reflective component-based middleware frameworks such as OpenCOM have also been proposed to address issues related to heterogeneity, adaptivity, and dynamic reconfiguration.

7. CONCLUSIONS

As AOSD gains in popularity, AO concepts are being applied to an ever-increasing range of domains from distributed computing to component-based software development. In this paper, we have shown how the Cactus system, which is oriented towards building configurable communication and middleware services, shares many design principles with AOSD systems. Based on our experience with Cactus and the customizable services that have been implemented using the system, we highlighted a number of useful features that are derived from its micro-protocol architecture and event-based execution model. While some of these features have counterparts in AO platforms, others are notably absent. We hope that our experiences will encourage the use, and introduction when needed, of these features into AOSD as the area continues to evolve to address issues in domains more traditionally associated with protocol frameworks such as Cactus.

Acknowledgments

The authors would like to thank F. Jahanian for pointing out the similarities between Cactus and aspect-oriented programming, and A. Rashid for his comprehensive overview of the area. Taïani has been supported in part by EPSRC project EP/C010345/1 “The Divergent Grid”.

8. REFERENCES

[1] G. Agha, S. Frolund, R. Panwar, and D. Sturman. A linguistic framework for dynamic composition of dependability protocols. In *Proc. Dependable Comp. for Critical Applications (DCCA)*, pages 197–207, 1992.

[2] R. Ȧlberg, J. Lawall, M. Südholt, G. Muller, and A.-F. L. Meur. On the automatic evolution of an OS kernel using temporal logic and AOP. In *Proc. Automated Software Engineering (ASE)*, pages 196–204, Oct 2003.

[3] D. Batory, J. Barnett, J. Garza, K. Smith, K. Tsukuda, B. Twichell, and T. Wise. GENESIS: An extensible database management system. *IEEE Trans. on Software Engineering*, SE-14(11):1711–1729, Nov 1988.

[4] N. Bhatti, M. Hiltunen, R. Schlichting, and W. Chiu. Coyote: A system for constructing fine-grain configurable communication services. *ACM Trans. on Computer Systems*, 16(4):321–366, Nov 1998.

[5] C. Bockisch, M. Haupt, M. Mezini, and K. Ostermann. Virtual machine support for dynamic join points. In *Proc. Aspect-Oriented Software Development (AOSD)*, pages 83–92, Mar 2004.

[6] J. Brichau and M. H. (editors). Survey of aspect-oriented languages and execution models. Tech. Rep. AOSD-Europe-VUB-01, AOSD-Europe, May 2005.

[7] B. Burke and M. Fleury. A killer app for AOP. *Linux Magazine*, 6(6):32, Jun 2004.

[8] W.-K. Chen, M. Hiltunen, and R. Schlichting. Constructing adaptive software in distributed systems. In *Proc. Distributed Computing Systems (ICDCS)*, pages 635–643, Apr 2001.

[9] T. Cohen and J. Gil. AspectJ2EE = AOP + J2EE: Towards an aspect based, programmable and extensible middleware framework. In *Proc. European Conf. on Object-Oriented Programming (ECOOP)*, pages 219–243, 2004.

[10] A. Colyer. AOP@Work: Introducing AspectJ 5. http://www.ibm.com/developerworks/java/library/-j-aopwork8,Jul 2005.

[11] T. Cottenier and T. Elrad. Contextual pointcut expressions for dynamic service customization. In *Proc. Dynamic Aspects Workshop (DAW)*, pages 95–99, Mar 2005.

[12] R. Das, M. Hiltunen, and R. Schlichting. Supporting configurability and real time in RTD channels. *Software: Practice and Experience*, 31(12):1183–1209, Oct 2001.

[13] Y. Deswarte, J.-C. Fabre, J.-M. Fray, D. Powell, and P.-G. Ranea. Saturne: A distributed computing system which tolerates faults and intrusions. In *Proc. Workshop on Future Trends of Distributed Computing Systems in the 1990's*, pages 329–338, Sep 1988.

[14] R. Douence, P. Fradet, and M. Südholt. Composition, reuse and interaction analysis of stateful aspects. In *Proc. Aspect-Oriented Software Development (AOSD)*, pages 141–150, Mar 2004.

[15] R. Filman, T. Elrad, S. Clarke, and M. Aksit. *Aspect-Oriented Software Development*. Addison Wesley, 2004.

[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[17] B. Garbinato, R. Guerraoui, and K. Mazouni. Implementation of the GARF replicated objects platform. *Distributed Systems Engineering Journal*, 2(1):14–27, 1995.

[18] J. He, M. Hiltunen, M. Rajagopalan, and R. Schlichting. QoS customization in distributed object systems. *Software: Practice and Experience*, (33):295–320, 2003.

[19] J. He, M. Hiltunen, and R. Schlichting. Customizing dependability attributes for mobile service platforms. In *Proc. Dependable Systems and Networks (DSN)*, pages 617–626, Jun 2004.

[20] J. Heidemann and G. Popek. Performance of cache coherence in stackable filing. In *Proc. Symp. on Operating Systems Principles (SOSP)*, pages 127–142, Dec 1995.

[21] M. Hiltunen. Configuration management for highly-customizable software. *IEE Proceedings: Software*, 145(5):180–188, Oct 1998.

[22] M. Hiltunen and R. Schlichting. An approach to constructing modular fault-tolerant protocols. In *Proc. Symp. on Reliable Distributed Systems (SRDS)*, pages 105–114, Oct 1993.

[23] M. Hiltunen and R. Schlichting. A model for adaptive fault-tolerant systems. In K. Echtle, D. Hammer, and D. Powell, eds, *Proc. European Dependable Computing Conf. (EDCC) (LNCS 852)*, pages 3–20, Oct 1994.

[24] M. Hiltunen, R. Schlichting, X. Han, M. Cardozo, and R. Das. Real-time dependable channels: Customizing QoS attributes for distributed systems. *IEEE Trans. on Parallel and Distributed Systems*, 10(6):600–612, Jun 1999.

[25] M. Hiltunen, R. Schlichting, and C. Ugarte. Enhancing survivability of security services using redundancy. In *Proc. Dependable Systems and Networks (DSN)*, pages 173–182, Jul 2001.

[26] N. Hutchinson and L. Peterson. The *x*-kernel: An architecture for implementing network protocols. *IEEE Trans. on Software Engineering*, 17(1):64–76, Jan 1991.

[27] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In *Proc. European Conf. on Object-Oriented Programming (ECOOP)*, pages 327–353, 2001.

[28] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proc. European Conf. on Object-Oriented Programming (ECOOP) (LNCS 1241)*, pages 220–242, Jun 1997.

[29] M. Killijian, J. Fabre, J. Ruiz-García, and S. Shiba. A metaobject protocol for fault-tolerant CORBA applications. In *Proc. Symp. on Reliable Distributed Systems (SRDS)*, pages 127–134, 1998.

[30] N. Loughran, N. Parlavantzas, M. Pinto, L. F. Fernández, P. Sánchez, M. Webster, and A. Colyer. Survey of aspect-oriented middleware. Tech. Rep. AOSD-Europe-ULANC-10, AOSD-Europe, Jun 2005.

[31] S. Maffeis. Design and implementation of a configurable mixed-media file system. *Operating Systems Review*, 28(4):4–10, Oct 1994.

[32] M. Mezini and K. Ostermann. Conquering aspects with Caesar. In *Proc. Aspect-Oriented Software Development (AOSD)*, pages 90–100, 2003.

[33] S. Mishra, L. Peterson, and R. Schlichting. Experience with modularity in Consul. *Software Practice & Experience*, 23(10):1059–1075, Oct 1993.

[34] R. Pawlak, L. Seinturier, L. Duchien, G. Florin, F. Legond-Aubry, and L. Martelli. JAC: an aspect-based distributed dynamic framework. *Software: Practice and Experence*, 34(12):1119–1148, 2004.

[35] M. Pinto, L. Fuentes, and J. Troya. A dynamic component and aspect-oriented platform. *Comput. J.*, 48(4):401–420, 2005.

[36] A. Popovici, T. Gross, and G. Alonso. Dynamic weaving for aspect-oriented programming. In *Proc. Aspect-Oriented Software Development (AOSD)*, pages 141–147, 2002.

[37] M. Rajagopalan, S. Debray, M. Hiltunen, and R. Schlichting. Profile-directed optimization of event-based programs. In *Proc. Programming Language Design and Implementation (PLDI)*, pages 106–116, Jun 2002.

[38] R. v. Renesse, K. Birman, M. Hayden, A. Vaysburd, and D. Karr. Building adaptive systems using Ensemble. *Software: Practice and Experience*, 28(9):963–979, Jul 1998.

[39] R. v. Renesse, K. Birman, and S. Maffeis. Horus, a flexible group communication system. *Comm. of the ACM*, 39(4):76–83, Apr 1996.

[40] D. M. Ritchie. A stream input-output system. *AT&T Bell Labs Technical Journal*, 63(8):311–324, Oct 1984.

[41] D. Schmidt, D. Box, and T. Suda. ADAPTIVE: A dynamically assembled protocol transformation, integration, and evaluation environment. *Concurrency: Practice and Experience*, 5(4):269–286, Jun 1993.

[42] S. Shrivastava, G. Dixon, and G. Parrington. An overview of the Arjuna distributed programming system. *IEEE Software*, 8(1):66–73, Jan 1991.

[43] D. Suvée, W. Vanderperren, and V. Jonckers. JAsCo: an aspect-oriented approach tailored for component based software development. In *Proc. Aspect-Oriented Software Development (AOSD)*, pages 21–29, 2003.

[44] F. Taïani, J.-C. Fabre, and M.-O. Killijian. Towards implementing multi-layer reflection for fault-tolerance. In *Proc. Dependable Systems and Networks (DSN)*, pages 435–444, Jun 2003.

[45] P. Tarr, H. Ossher, and J. S. Sutton. Hyper/j: multi-dimensional separation of concerns for java. In *Proc. Software Engineering*, pages 689–690, 2002.

[46] G. Wong, M. Hiltunen, and R. Schlichting. A configurable and extensible transport protocol. In *Proc. IEEE Communications and Computer Societies (INFOCOM)*, pages 319–328, Apr 2001.