An Extensible Binding Framework for Component-Based Middleware

Nikos Parlavantzas, Geoff Coulson, and Gordon S. Blair
Distributed Multimedia Research Group, Dept. of Computing, Lancaster University, UK
[parlavan, geoff, gordon]@comp.lancs.ac.uk

Abstract

One of the most significant limitations of current middleware platforms, both commercial and research, is that they typically support only a small, pre-defined, set of fundamental binding types (e.g., remote method invocation). This restriction limits the scope of platforms in that they cannot easily accommodate, or easily be extended to accommodate, richer or more specialised forms of interaction (e.g. events, media streaming, multicast, and many others discussed in the paper). This paper describes a highly extensible, component-oriented framework for the definition and implementation of such binding types. We motivate and specify the framework in detail and evaluate it by providing examples of its use.

1. Introduction

A primary function of middleware is to interconnect application components. To this end, middleware platforms offer interaction abstractions called bindings. However, one of the most significant limitations of current middleware platforms, both commercial and research, is that they typically support only a small, pre-defined, set of fundamental binding types (e.g., remote method invocation). This restriction limits the scope of platforms in that they cannot easily accommodate, or easily be extended to accommodate, richer or more specialised forms of interaction (e.g. events, media streaming, or message queuing). Furthermore, when multiple binding types are indeed supported, they tend to be implemented in an ad-hoc way and to rely on distinct infrastructures. For example, CORBA events [16] and media streams [17] have completely separate APIs and implementations and, moreover, these are completely distinct from the API/ implementation of the core remote method invocation binding type. Such a lack of integration leads to missed opportunities for design and code reuse, increases the cognitive load on middleware users, who have to deal with multiple different APIs, and leads to problems in realising globally-coordinated QoS across binding types.

To address these concerns, this paper describes a highly extensible and flexible component-based framework for the design, development, deployment and use of binding types. By capturing diverse forms of interaction as binding types (hereafter, 'BTs') within the framework, we argue that our approach can significantly simplify application development, increase the interoperability options available to developers, and promote the reuse of recurring interaction patterns and mechanisms.

Here are some examples of BTs that we would like to be able to support: (1) remote method invocation (RMI) in its numerous variants, (2) messaging and eventing in their numerous variants, such as asynchronous method invocation, message queuing and publish/subscribe models, (3) continuous media streaming, (4) group communication in its numerous variants, (5) shared data spaces for communication such as tuple spaces, blackboard systems, or mailboxes, (6) SQL links between applications and databases, (7) FTP links, (8) Unix-like pipes, (9) BTs that encapsulate voting protocols, auction protocols, (10) distributed resource allocation protocols, (11) BTs that execute a workflow process involving multiple processing entities, (12) BTs that encapsulate common interactions in e-Science GRIDs, (13) drag-and-drop protocols between GUI components, (14) Model-View-Controller collaborations, or (15) multi-player game protocols.

Such communication (and, indeed, coordination) services are needed in many different contexts by a variety of applications. Therefore, providing these services as part of the middleware is highly advantageous. Of course, it is always possible to implement such facilities in terms of the small, fixed set of BTs offered by current platforms. But the purpose of middleware is not to provide a theoretically minimum set of communication primitives; rather, it should facilitate the development of enterprise applications by raising the level of abstraction over interaction mechanisms. Platforms should therefore offer an (extensible) range of

BTs, while ensuring that BT APIs are consistent and that their implementations are efficiently integrated within the platform.

The specific goals of this paper are:

- to provide a detailed overview of our extensible binding framework in terms of the design, development, deployment and use of BTs
- to illustrate how the binding framework can be used to define three representative BTs.

The paper is structured as follows. First, section 2 provides context for the framework in terms of the OpenCOM/ OpenORB middleware technology on which it is founded, the assumed conceptual binding model, and the way in which binding types are specified abstractly. Then, sections 3, 4, and 5 present the binding framework in detail, and section 6 illustrates how it can be used to construct three representative BTs. Finally, section 7 reviews related work and section 8 evaluates the framework and draws conclusions.

2. Context

2.1. OpenCOM and OpenORB

The binding framework builds on our previous work on the OpenCOM [4] component model and the OpenORB component-based middleware architecture [5,19]. OpenCOM is a lightweight, non-distributed component model inspired by Microsoft COM [14]. It is designed to include only aspects that are essential in supporting the notion of a component and can be used for composing both applications and middleware. The OpenORB architecture supports the development of highly configurable and dynamically reconfigurable reflective middleware platforms and is structured in terms of *component frameworks* [24]. Essentially, component frameworks are reusable architectures that apply to specific domains and are designed to be instantiated in terms of components. The binding framework that is the subject of this paper is specified and implemented as such a component framework. In general, the role of component frameworks (hereafter 'CFs') is to provide rules for structuring localised domains of middleware functionality (e.g. concurrency support, buffer management, message demultiplexing, or pluggable protocols). Typically, CFs include software that supports the rules at runtime and helps maintain integrity in the face of dynamic reconfiguration. For example, the pluggable protocols CF defines rules for composing "plugged-in" protocol components and manages their dynamic reconfiguration.

The CF-based structure of OpenORB, complete with the binding framework (labelled 'Binding CF'), is illustrated in Fig. 1. As implied by Fig. 1, the binding framework has access, in the 'communications layer', to an extensible range of CFs—the diagram shows a pluggable protocols CF and a media streaming CF. In addition, it has access to a range of 'resources layer' CFs—either indirectly, via communications layer intermediaries, or directly if desired. Note also that the three layers are themselves encapsulated by an overall top-level CF called 'Middleware Top CF'. This is responsible for managing the lifecycle of its encapsulated CFs, for enforcing policies concerning dynamic changes in layer composition, and for supporting service discovery to resolve dynamic dependencies between layers [5]. Moreover, it imposes that the encapsulated CFs conform to a 'resources framework' (not discussed in this paper) that allows fine-grained control over, and accounting for, middleware-managed resources like threads, buffers and network bandwidth (e.g. to simplify QoS management) [20].

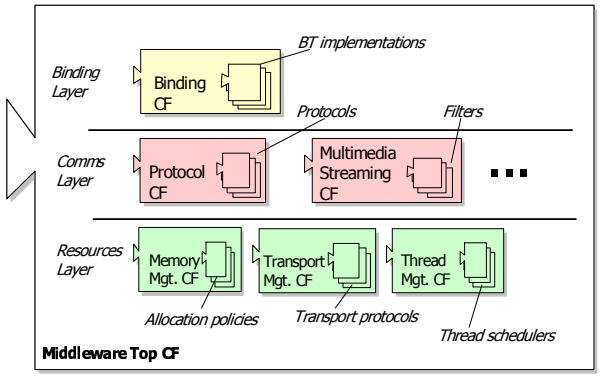

Figure 1. The OpenORB architecture

2.2. Conceptual Binding Model

The binding framework is based on a conceptual binding model, influenced by RM-ODP [13], which describes what bindings are, how they are established and controlled, what entities are involved, and how those entities interact. The key properties of this model, which are mainly determined by the fact that it must be very *generally applicable*, are as follows: First, the model encompasses both *local* bindings which are primitively realised within a single address space, and *distributed* bindings, which can span address spaces and machines. Second, the model assumes *multi-party* bindings—i.e., bindings between any number of participants (an RMI binding involving two participants is merely a special case of this more general case). Third, the model

supports *explicit* binding—i.e., bindings are created explicitly by application code and are themselves represented as components. This 'reification' of bindings allows applications to select from a possible range of binding types, and opens the possibility of controlling, managing and adapting bindings at runtime (via interfaces on the binding components). Finally, the model supports *third-party binding*. This means that the party that initiates binding establishment may or may not itself be a communicating participant in the binding. Aside from reasons of generality, third-party binding is particularly beneficial because it isolates the component interconnection logic, thus making it easier to change.

At a more detailed level, the conceptual binding model defines a number of basic entities: viz. *participants*, *binders*, *bindingCtls*, *generators*, *reps*, *irefs*, *resolvers*, and *APUs*. These entities, together with their inter-relationships, are illustrated in Fig. 2. In brief, bindings are established between binding participants and the responsibility for binding establishment is assigned to binders. Binders take as input a number of components representing participants, together with related information such as QoS specifications. They then verify that the supplied participant components conform to appropriate participant roles, which are defined in the associated BT specification (see section 2.3). Subsequently, binders invoke appropriate operations on the participant components and establish the binding with the aid of services offered by the underlying platform. If binding establishment succeeds, the binder returns a component (*bindingCtl*) through which the binding can be controlled and managed.

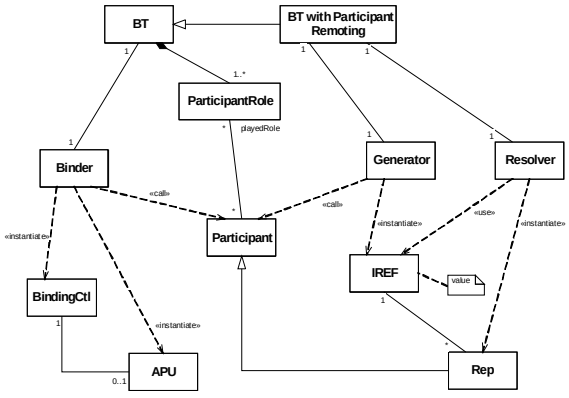

Figure 2. The binding model entities

Participants that are remote with respect to a binder's location are represented by *reps* ('remote participant representatives'). The process of creating a rep falls into two stages as follows. First, a *generator* is used at the participant's (remote) site to generate both an *iref* and an associated communication infrastructure. An iref is a value that represents a participant and can be passed around the distributed system. Second, the iref is transferred to the binder's site (by some means or other) at which it is passed to a *resolver* that is responsible for creating a corresponding rep. This whole process is referred to as *participant remoting*.

A very common special case, which is specifically accommodated by the framework, involves first-party bindings that are initiated by an anonymous participant (e.g., the 'client' in traditional RMI bindings). The essence of such bindings is that one participant, the binding initiator, is not explicitly represented; its properties are implied because it is collocated with the binder. In this case, when the binding is established, the binder returns a so-called *APU* ("anonymous participant use") component (e.g., the 'proxy' in traditional RMI bindings). Essentially, the need for *APUs* is a consequence of the asymmetric nature of object interfaces.

Finally, it is important to note that all of the above defined entities (resolvers, binders, etc.) are in fact *roles*, and that individual components can play more than one role at a time (e.g., a resolver component can also serve as a binder). A corollary of this is that a single component can take part (i.e., play the participant role) in many different bindings of different types. This reduces the coupling of interacting components to the used BTs, which increases the reusability of components and allows the system to be easily adapted by evolving or replacing BTs.

2.3. Specifying Binding Types

The framework defines BTs as systems that enable components to cooperate in specific ways. Since the goal of BTs is to support and mediate a given scope of interactions, a BT specification can most naturally be expressed using *collaborations* as defined in UML [18]. More specifically, a BT specification can be decomposed into four types of collaborations between the BT and two types of external role: *binding participants* and *binding managers*. Binding participants are components that interact through BT-provided bindings and binding managers are components that establish and control bindings. The collaborations are characterised as follows:

- *Binding Participation* describes the interaction among binding participants that is supported by and embodies the purpose of the BT.
- *Iref Generation* and *Iref Resolution* describe the process of managing irefs, which is a prerequisite to establishing bindings with remote participants.

- *Binding Establishment* describes the required sequence of actions (initiated by the binding manager role) to set-up a binding between some number of participants.
- *Binding Control* describes the process of managing (again, by the binding manager role) an already-established binding, involving tasks such as, e.g. monitoring and adaptation, controlling and changing the QoS, adding/ removing participants, and binding destruction.

Whereas the first collaboration, binding participation, is unspecified by the binding framework and can take any required form, the remainder are all constrained by the need to conform to the Binding API contract described in section 4. These properties together facilitate the creation of new BT specifications, while minimizing any restrictions on the range of BT-supported interactions. From a practical point of view, we use UML tools and techniques to specify both functional and extra-functional properties of the BT collaborations. Of course, specialized QoS modelling languages expressed as UML extensions could also be applied if needed (e.g., [1]).

3. Overall Architecture of the Binding CF

The Binding CF provides abstractions and rules that support both the specification (according to the scheme outlined in section 2.3) and the implementation of BTs. These abstractions and rules are designed to be highly generic to maximise the diversity of useful communication and coordination mechanisms that can be captured as BTs.

Architecturally, the Binding CF defines *three* roles that are played by participating components: i) the *binding user* role (this combines the binding manager/ participant roles specified in section 2.3, ii) the *BT implementation* role, and iii) the *Binding CF implementation* role. The CF itself is decomposed into two contracts which structure the cooperation between the three roles (see Fig. 3):

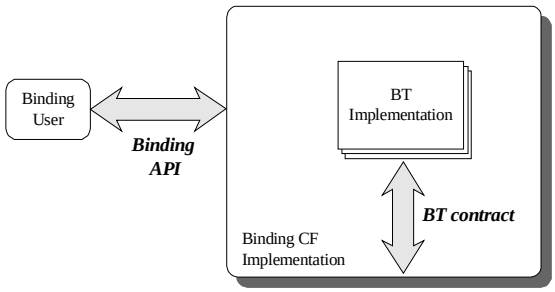

Figure 3. The two binding CF contracts

- The *Binding API* defines the view of BTs seen by binding users and provides the programming model for handling bindings.
- The *BT Contract* defines the collaboration between BT implementations and the Binding CF implementation, which has the goals of (1) facilitating the development of BT implementations, and (2) supporting their dynamic configuration.

All of these issues are dealt with in detail in sections 4 and 5 below.

4. The Binding API

The Binding API provides generic interfaces corresponding to the conceptual binding model entities that were described in section 2.2, together with rules governing their BT-specific extension. All BT specifications (and thus implementations) must conform to the Binding API contract—i.e. they can extend the contract as required but must include the generic interfaces discussed in this section.

4.1. Realising the Basic Binding Model Entities

4.1.1. Preamble. BTs are identified by globally unique names, *BT identifiers*, that are mapped to deployable (sets of) components by the Binding CF. This separation of BT specification from implementation, coupled with the fact that BT specifications are assumed to be immutable, enables BT implementations to be upgraded without affecting binding users.

The *BindingCFI* component, which represents the Binding CF implementation, provides the initial access point for BT-provided facilities. It offers the interface *IBTAccess* that provides three operations which respectively resolve BT identifiers to binders, resolvers and generators (see below[1]).

```
interface IBTAccess : IUnknown {
   HRESULT GetBinderForBT ([in] OLECHAR* BTid,
    [in] REFIID riid,
      [out] IUnknown**);
   HRESULT GetResolverForBT ([in] OLECHAR*
    BTid,[out]IResolver**);
   HRESULT GetGeneratorForBT ([in] OLECHAR*
    BTid,[in] REFIID riid,
```

[1] *IUnknown*, as defined by Microsoft's COM, is a special interface supported by all components. It includes an operation, *QueryInterface()*, that allows one to dynamically discover other interfaces supported by the component. REFIID represents an interface type and is typically used to request a specific interface on a component in a single step.

```
    [out] IUnknown**);
}
```

4.1.2. Binder. The Binding API mandates that there is a single binder per supported BT at runtime. The component type (i.e., set of supported interfaces) of binders is highly dependent on the associated BT. The Binding API itself requires only that a single, generic interface, *IGenericBinder*, must be minimally offered. The API further recommends that other BT-specific interfaces follow the general form of *IGenericBinder* for consistency reasons (where possible).

```
interface IGenericBinder : IUnknown {
   HRESULT Bind ([in] IUnknown* participants,
    [in] long participantCount,
    [in] VARIANT bindContext,
    [out] BindingCtl** ppBindingCtl,
    [out] IAPU** ppAPU);
}
```

The *Bind()* operation takes as arguments a set of participants (represented as *IUnknown* pointers) and a *bindContext* value (of *VARIANT* type) that holds generic context information. Using this information, *Bind()* attempts to establish a binding, and, on success, it returns a *bindingCtl* and (optionally) an *APU*. The interpretation of the bindContext parameter is BT-specific; for example, it can be used to pass a QoS specification in an agreed textual format, or a reference to a QoS negotiator component.

Despite its flexibility, *IGenericBinder* cannot possibly capture all the potential binding establishment scenarios, which means that additional, BT-specific binder interfaces will normally be required. However, the benefit of generic interfaces, such as *IGenericBinder*, is twofold. First, they largely increase the scope of binding scenarios that can be accommodated, while minimally impacting ease of use. Second, they act as uniform interfaces, which promotes composability. For instance, they could be used by an automated, metadata-driven tool that receives as input an architectural description of a system in terms of components and connectors and instantiates the system.

4.1.3. Generator. Generators are responsible for creating irefs and their associated communication infrastructure. There is a single generator per BT, and the component type of the generator is specific to that BT. The API defines only a single, generic interface, *IGenericGenerator*, which must be minimally offered. This provides an operation *Generate()* that takes as arguments a participant, an interface type and a BT-specific *generateContext* value (of *VARIANT* type); it returns an iref.

The *StdGenerator* component is a special generator implementation, provided by the framework by default, that is used to marshal arbitrary component references (i.e., pointers to component instances). Built-in support for marshalling component references is necessary because the component model presumes object-oriented interfaces, which means that component references may need to be passed as arguments over already established bindings. The *StdGenerator* is accessed via *BindingCFI* by using a well-known BT identifier. It offers the *IGenericGenerator* interface and implements it by selecting the target BT (see below), finding the corresponding BT-specific generator (via the *BindingCFI*) and delegating the generation request to it. The BT selection step gives an opportunity for the component to itself decide a preferred BT for iref generation (marshalling). Specifically, if the component implements the *IStdMarshal* interface, *BindingCFI* invokes it to retrieve a BT identifier; otherwise a default BT is used.

4.1.4. Resolver. Resolvers offer the interface *IResolver* which has a single operation *Resolve()* that takes as arguments an iref and an interface type, and returns a reference of the requested type. There is a single resolver per BT.

The *StdResolver* component is a special resolver implementation, provided by the framework as default, that can resolve arbitrary irefs. *StdResolver* implements the interface *IResolver* and is accessed through the *BindingCFI* via a well-known BT identifier. It works by extracting the BT identifier from the iref (exploiting the standardised iref format—see below), finding the associated resolver (via the *BindingCFI*), invoking the resolver and returning the resulting reference.

4.1.5. Representations of the other Basic Entities. Participants are represented in the Binding API as components that are able to conform to some participant role defined by the associated BT specification. Binders and generators are expected to dynamically verify that participant components conform to the expected role by using *IUnknown.QueryInterface()* and/or extended meta-information provided by the component model. For example, a binder (or generator) can check that a participant offers and uses a given set of interfaces. Alternatively, it can check that an offered interface is 'one-way' (i.e., its operations have only input parameters). This dynamic verification allows components to participate in various BTs—even BTs that did not exist when the components were implemented—and play various roles. Moreover, components can be upgraded without affecting their BT participation.

Irefs are represented as strings of a standardised format. Each iref is associated with a single BT and

contains a BT identifier together with BT-specific data. *Reps* are represented as components that support interfaces used by binders for the purpose of binding establishment. A rep is associated with a single BT, which defines its component type. At a minimum, reps must offer the interface *IRep* which defines a single operation to return the rep's associated BT identifier.

APUs are represented as components that minimally offer the interface *IAPU*, which defines an operation to return the associated *bindingCtl*. Finally, *bindingCtls* are represented as components that minimally offer the generic interface *IBindingCtl* with operations to destroy the binding, get/set the QoS of the binding (represented generically as a string) and subscribe/unsubscribe to events generated by the binding. *BindingCtl* components will normally offer additional, more sophisticated and strongly-typed interfaces.

4.2. Collaborations specified by the Binding API

We are now in a position to illustrate the use of the Binding API by describing the interactional view of the generic collaborations discussed in section 2.3.

Binding Establishment. The steps for establishing a binding are:

1. The binding user selects a BT and invokes *BindingCFI::IBTAccess* to retrieve a binder interface (e.g., *IGenericBinder*).
2. The binding user invokes the binder interface, passing the participant components and other required information as arguments.
3. The binder establishes the binding, and returns a *bindingCtl* and (optionally) an *APU*.

Iref Generation. The steps for generating an iref are:

1. The binding user selects a BT and invokes *BindingCFI::IBTAccess* to retrieve a generator interface. The BT must support participant remoting.
2. The binding user invokes the generator interface, passing as arguments a participant component and other information.
3. The generator generates an iref and associated infrastructure (e.g., stub, protocols) and returns the iref to the binding user.
4. The binding user exports the iref (e.g., to a naming service, a trader, a web page).

The API recognises a specialisation of this collaboration that applies when the binding user has no context in which to base a selection of BT. This situation typically arises when a component reference must be passed through an already established binding. In this case, the binding user (e.g., the part of the already established binding that performs marshaling) invokes the *IGenericGenerator* interface on the default *StdGenerator* component, which in turn selects a more specific generator and delegates the invocation to it, as seen in section 4.1.3.

Iref Resolution. An iref may arrive through an already established binding or be obtained from a file, the GUI etc. The steps for resolving an iref are:

1. The binding user selects the *StdResolver* BT identifier and retrieves the *IResolver* interface (using *BindingCFI::IBTAccess*).
2. The binding user invokes *StdResolver::IResolver.Resolve(),* passing the iref as the argument.
3. *StdResolver* extracts the BT from the iref, retrieves the *IResolver* on the corresponding resolver (using *BindingCFI::IBTAccess*), invokes it, and returns the result to the binding user.

There are two possible BT-dependent variations regarding the resulting component: i) the resulting component is a rep, which can subsequently participate in binding establishment, and ii) the resulting component is an *APU* that is associated with a *bindingCtl*. In the second variation, the resolver implementation also plays the role of the binder and completes the binding between the participant corresponding to the iref and an anonymous participant.

Binding Control. The binding user controls a binding through the *bindingCtl* that is obtained as a result from either binding establishment or iref resolution, as described above.

5. The BT Contract

The BT Contract defines the collaboration between BT implementation components and the Binding CF implementation itself. A structural view of the collaboration is shown in Fig. 4. The interfaces and constraints pertaining to each party are summarized next.

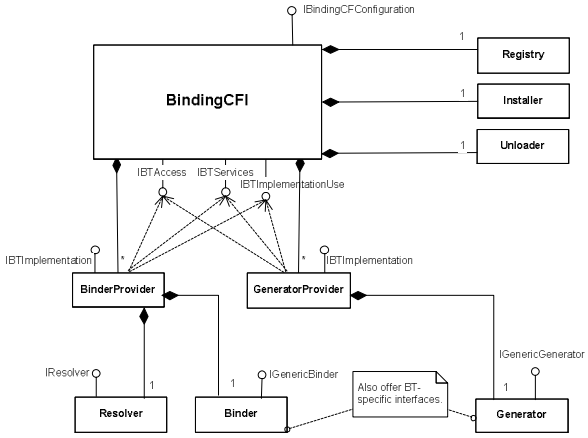

Figure 4. Structure diagram for BT contract

5.1. BT implementations

The BT Contract specifies that the implementation of a BT is packaged into two separate singleton components: a *BinderProvider* component and a *GeneratorProvider* component. The former supplies implementations for the binder and resolver roles of the BT; the latter supplies the generator role. The motivation for separating the BT implementation into two distinguished components is that the two parts are largely self-contained and frequently only a single part needs to be used within a process (e.g., an RMI generator would not be useful in the client tier of typical enterprise applications). The separation means that the two parts can be deployed and replaced independently. Of course, the BT components can make use of, and be composed of, further components; the *BinderProvider* and *GeneratorProvider* are simply the components that are recognised by the BT Contract.

The two mandatory BT implementations, *StdResolver* and *StdGenerator* (see section 4.1), are packaged as normal BT components and thus they are replaceable. For example, *StdResolver* can be replaced to modify the standardised iref format, and *StdGenerator* can be replaced to modify the BT selection process.

Both types of BT component offer an *IBTImplementation* interface which defines operations for lifecycle management (i.e., initialisation and termination) and for retrieving the 'state' of the component. A BT component may be in one of two states: i) READY, which indicates that the component is not currently being used by any clients, and ii) ACTIVE, which means that the component is currently in use (i.e. clients hold references to it). The component moves between the ACTIVE and READY states autonomously (exploiting reference counting) and can be terminated only when it is in the READY state. Moreover, the component is responsible for notifying the Binding CF implementation about its state changes (see below).

A BT component can have two kinds of usage dependencies: i) dependencies on other BTs (e.g., a BT component that realises a distributed auction protocol—see section 6.4—may rely on an RMI BT), and ii) dependencies on services provided by the lower platform layers (i.e., the communications and resources layers, see section 2). To resolve such dependencies, BT components use the *IBTAccess* and *IBTServices* interfaces respectively, both of which are offered by the Binding CF implementation (see below).

5.2. The Binding CF Implementation

The Binding CF implementation is packaged within a singleton component (*BindingCFI*) which has the following three responsibilities: i) to act as the access point for BTs, ii) to manage the configuration of BT components, and iii) to provide BT components with access to other BTs and low-level services.

To carry out the first responsibility, *BindingCFI* exposes the *IBTAccess* interface, as was described in section 4.1, which it implements internally by invoking a *registry* component. This maintains a shared, per-node, persistent repository that maps from BT identifiers to *BinderProvider* and *GeneratorProvider* component identifiers. If a BT identifier cannot be found in the per-node repository, *BindingCFI* uses an *installer* component. This has the responsibility to contact some remote source (e.g., a global name service, or a specific URL) and download and install the components corresponding to the globally unique BT identifier. After finding suitable component identifiers, *BindingCFI* instantiates, initialises, and uses the corresponding BT component instance to retrieve the interface requested by the *IBTAccess* invocation. The instance is kept in an internal registry for later reference.

To accomplish its second responsibility, *BindingCFI* uses the *IBTImplementation* interface (see section 5.1) to cause BT components to initialise and terminate themselves. Moreover, it offers the *IBTImplementationUse* interface to accept notifications about changes in their state. Different lifecycle management policies are implemented using pluggable *unloader* components, which track state changes in BT components and decide when to remove them. Moreover, as the standard pattern in OpenORB mandates [5], *BindingCFI* exposes a reconfiguration interface (*IBindingCFConfiguration*) that allows clients to dynamically add, remove, replace and retrieve

BinderProvider, *GeneratorProvider* and unloader components. The reconfiguration interface disallows the removal of active BT components.

To carry out its third responsibility, *BindingCFI* exposes the interfaces *IBTAccess* and *IBTService*. *IBTAccess* we have already discussed in section 4.1.1; *IBTServices* defines an operation *GetService()* which receives as an input a service identifier and an interface type and returns an interface of the requested type.

```
Interface IBTServices : IUnknown {
   HRESULT GetService([in] OLECHAR* serviceID,
    [in] REFIID riid,
      [out] IUnknown**);
}
```

The set of available service identifiers, associated interfaces and interface contracts is standardized by the top-level CF in the middleware architecture (see Fig. 1). This minimizes the dependencies of the binding CF on its context in the architecture and increases its reusability.

6. Some Example BTs

6.1. Overview

We now demonstrate the extensibility and ease of use of the Binding CF by presenting the design and outline implementation of three BTs: a simple RMII BT, a publish/subscribe BT and an auction protocol BT. Space constraints restrict us to consideration of just three BTs; the implementation of a wide range of others is discussed in [25].

Adding a new BT requires two separate pieces of development work: i) a BT specification, which must conform to the Binding API, and ii) a BT implementation, which realises the specification and conforms to the BT Contract. BT specifications, which are discussed in terms of the collaborations discussed in section 2.3, are the public part that is necessary both for providing new implementations, and for documentation purposes. In the following sections, each example BT is specified using the following format: (1) *Participant Roles*—describes participant roles and their relationships, (2) *Binding Participation*—describes how participants interact through an established binding, (3) *Iref Generation/Resolution*—describes the management of irefs, (4) *Binding Establishment*—describes the process of binding establishment, (5) *Implementation*—outlines possible implementations of the BT, and also discusses how the API could accommodate more sophisticated extensions or variations of the BT.

6.2. A Remote Method Invocation BT

This BT provides the traditional RMI interaction style and can easily be accommodated by the framework with very minimal extensions to the Binding API.

Participant Roles

- *Server*—a component that accepts remote method invocations.
- *Client*—an anonymous participant that invokes operations on the server.

Binding Participation. Client-originated invocations result in corresponding server invocations. The communication is synchronous and the delivery guarantee is at most-once; other non-functional properties, such as response delay, are not constrained.

Iref Generation/Resolution. The RMI generator and resolver offer the standard, generic interfaces defined in the API.

Binding Establishment. The resolver plays also the role of a binder; it establishes the binding using the iref and returns an *APU* component, which serves as the traditional proxy to the server.

Implementation. The implementation builds on the low-level services provided by OpenORB. The API can also accommodate more sophisticated extensions of this BT, which offer flexible bindings with QoS support. In such a BT, the *bindingCtl* component, exposed through the *APU*, would be used to monitor and adapt the binding at the client-side (e.g., receive events notifying a drop in throughput). At the server-side, the generator would offer extra interfaces to configure its behaviour (e.g., to decide which protocol stack to use for the server). Moreover, the resolver could create a rep that needs to be explicitly bound using a binder. The separate binding establishment step could support negotiation of QoS properties of the bindings.

6.3. A Publish/ Subscribe BT

This BT provides a publish/ subscribe interaction style, whereby publishers and subscribers are indirectly associated through a separate entity, termed an ‘event channel’. The event channel functionality is realised by the BT. An ‘event’ is a single invocation on a event interface, originated by a publisher and delivered by the BT to the appropriate subscribers.

Participant Roles

- *Publisher*—an anonymous participant that originates events (see Fig. 5).
- *Subscriber*—a component that offers the event interface and receives events.

- *Event Channel*—a logical participant realised by the BT itself. It is associated with a single event interface, which must be one-way; that is, all operations must contain only input parameters.

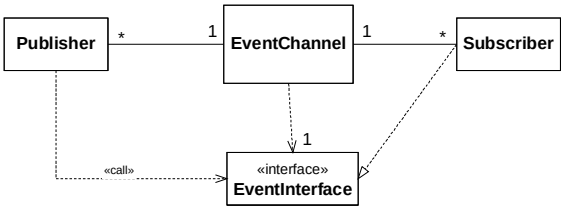

Figure 5. Publish/ Subscribe BT participants

Binding Participation. When a publisher invokes an operation on an event interface, the corresponding operation with the same arguments gets invoked on all subscribers. The delivery guarantee is at-most-once, and event firing is synchronous; the publisher is blocked until the invocations on the subscribers have been completed. The BT does not specify the order or any further QoS characteristics of event delivery.

Iref Generation/Resolution. Irefs are generated for the event channel and subscriber participants—publishers are anonymous. The standard generator API is extended to offer two extra interfaces: *IEventChannelGenerator*, and *ISubscriberGenerator*. The *Generate()* operation of the first receives the event interface type as an argument, verifies that this is 'one-way' using reflection, and creates the channel iref. The *Generate()* operation of the second accepts a subscriber interface pointer, verifies that the interface is one-way, and creates the iref. The channel/ subscriber irefs are transformed into corresponding rep components through the resolver.

Binding Establishment. This collaboration is separated into two parts: i) the binding of publishers to the event channel, and ii) the binding of subscribers to the event channel. The standard binder API is extended to offer two extra interfaces: *IPublisherToChanBinder*, and *ISubscriberToChanBinder*. The *Bind()* operation of the first receives as argument a channel rep and returns both a *bindingCtl* and an *APU*. The *APU* supports the event interface and can be used for firing events. This separate publisher-to-channel binding step can be bypassed in this simple BT. Indeed, it can be assumed that the channel iref has all the necessary information to enable binding establishment at iref resolution time (e.g., when an iref enters an address space). In other words, the channel rep implements the event interface and can be used 'as-is' for firing events (i.e., the resolver serves also as a binder). The *Bind()* operation of *ISubscriberToChanBinder* accepts a channel rep and subscriber and returns a *bindingCtl* that represents the subscription. The subscriber component can either be a subscriber rep associated with the channel's event interface or any local object supporting the event interface (verified dynamically). In other words, subscribers may not need to be "remoted" in applications of this BT (i.e., there is no need to generate/ resolve a subscriber iref). The subscriber could even be a proxy to a remote component produced by the RMI BT.

Implementation. A likely implementation could be based on a simple RMI BT. The event channel is reified by a channel manager component, which maintains the current set of subscribers. The publishers invoke the channel manager using the RMI BT and this forwards the invocation, again through RMI, to all the associated subscribers. Instead of this centralised implementation, another possibility is to rely on a multi-party protocol implementation, provided as a low-level OpenORB service. The event channel would then be represented as a multicast address. Note that binding users are always isolated from the details of the implementation (e.g., the existence of the channel manager component). A still more sophisticated extension of this BT could enable control of the QoS characteristics of event delivery. In such a BT, the two kinds of *bindingCtl* as well as additional generator interfaces could be used to configure and negotiate QoS properties, such as reliability, priority and ordering of events.

6.4. An Auction BT

This BT supports the realisation of an auction protocol, which mediates resource exchange and corresponding payment between a number of agents. BTs for auctions could be useful as part of middleware for electronic commerce applications [25] or for any other area requiring market-based resource allocation mechanisms.

The supported auction protocol is a variation of the common English 'open-outcry' auction type. In this protocol, the bid price is continuously increasing, and potential buyers have a certain amount of time to indicate their willingness to buy at the current price. The auction continues until no buyers are prepared to pay the proposed price. The buyer that first accepted the last bid price is the winner of the auction. The seller can set a minimum selling price (reserve), below which there is no sale.

The auction protocol is encapsulated and driven by the BT; this results in simplifying and decoupling the buyer and seller roles. Each auction is configured with the following information: item description, seller contact

information, the initial price, bid increment, duration of each round, and reserve price.

Participant Roles

- *Seller*—a component that initiates an auction in order to sell some item. It uses the interface *ISellerUse* to start the auction and offers the interface *ISeller* to receive the auction result (see Fig. 6).
- *Buyer*—a component that participates in an auction wishing to buy the related item. It offers the interface *IBuyer* to receive the current bid price and the auction result. It uses the interface *IBuyerUse* to indicate acceptance of a bid price.

Binding Participation. This collaboration describes the interaction among seller and buyers through the BT-provided binding. A seller initiates the auction by asynchronously invoking the *ISellerUse.StartAuction()* operation. The BT announces a new bid price by asynchronously invoking *IBuyer.NewBid()* on each buyer. A buyer accepts the bid by invoking *IBuyUse.AcceptBid()* (also asynchronously). When the auction is completed, the BT issues the asynchronous invocation *Lost()* to the unsuccessful buyers and the synchronous invocations *Won()* and *AuctionCompleted()* to the winner (if there is one) and seller respectively. The *Won()* operation passes the contact details of the seller to the winner so that the sale can be arranged subsequently. The invocations are assumed to be delivered at-most-once. The sequence diagram in Fig. 7 illustrates an example scenario for the collaboration.

Iref Generation/Resolution. Irefs are generated for both buyers and sellers. The API generator component is extended to offer two extra interfaces: *ISellerGenerator*, and *IBuyerGenerator*. The *Generate()* method of the first receives as arguments an *ISeller* interface pointer and a structure with the auction configuration data (item description, seller contact details, initial price, bid increment, round duration, reserve price) and returns a seller iref. Furthermore, it passes an *ISellerUse* pointer to the seller (through the *ConnectSellerUse()* operation), which the seller can use to start the auction. The *IBuyerGenerator.Generate()* method accepts an *IBuyer* interface pointer, connects it with the *IBuyerUse*, and creates the buyer iref. The resolver transforms the buyer/seller irefs to corresponding rep components. The seller rep (*ISellerRep*) exposes information about the auction that is useful to binding initiators, namely the item description, initial price, bid increment and round duration (the seller details and reserve price are hidden).

Binding Establishment. This collaboration describes the binding of buyers to a seller. The auction binder offers the interface *IBuyerToSellerBinder* with the operation *Bind()* that receives as argument a seller rep and one or more buyers and returns a *bindingCtl*. The buyers are either buyer reps or local objects implementing *IBuyer*. The binding model enables multiple buyers from different sites to participate in the same auction. Note that after the auction is completed, the seller reps are invalidated and any attempt to bind them fails.

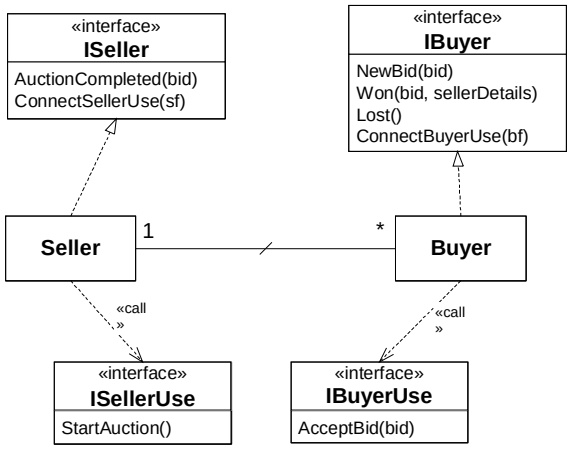

Figure 6. Auction BT participants

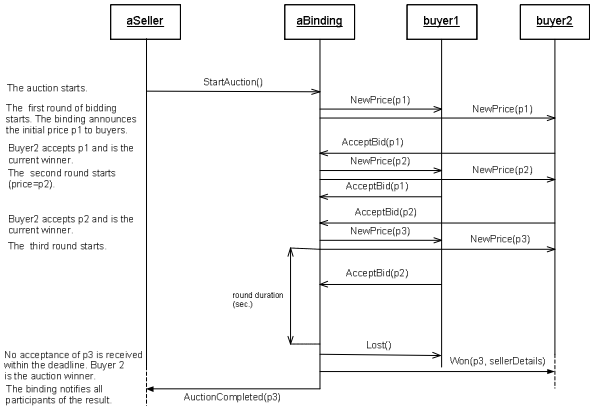

Figure 7. Auction Scenario

Implementation. The implementation of this BT could rely on an asynchronous RMI BT and/or protocol implementations provided by lower platform layers. Versions of the BT that allow further variation in auction configurations and even in the auction protocol itself can be accommodated by extending the generator and rep interfaces.

7. Related Work

CORBA services such as the event service [16] or the audio/video streams service [17] represent an attempt to provide different binding types within a single middleware architecture. However, this approach has many limitations. New 'binding types' are implemented in an ad-hoc way by exploiting non-portable lower-level infrastructure interfaces, and there is no coordination between the implementations of services, which is important if QoS is to be managed effectively. Furthermore, the services are not replaceable and their dependencies are not visible. Moreover, the programming models of the different services have little in common, which increases the cognitive load on middleware users. The same criticism applies to similar services from the Java world, such as the Java Message Service [23].

Enterprise component technologies, such as EJB, CORBA Component model and COM+/.NET simplify the use of middleware services by exposing a declarative programming model. However, these technologies do not address the need for multiple binding types; they only offer a fixed set of types that typically includes remote method invocation, messaging and events (e.g., [15]).

The RM-ODP standard [13] introduces the notion of binding as the locus of interaction between objects. RM-ODP assumes three kinds of interfaces, namely operational, signal and stream interfaces, and associated models of interaction. Both operational and stream interfaces can be defined in terms of signal interfaces. Given the fact that there are a multitude of useful interaction models and the chosen three are not orthogonal, the distinction seems to unnecessarily complicate the programming model. Despite this limitation, RM-ODP, by itself, does not restrict the possible communication structures between interacting objects and allows multiparty bindings even between interfaces of different kinds.

RM-ODP concepts are followed in many research platforms. For example, the ReTINA [7] project designed an ORB architecture featuring a binding framework based on those concepts. At the core of the framework lies a distinction between location-independent interface references, which are managed by the ORB kernel, and binding factories which define various ways to access and interact with interface references. Jonathan [9] is an ORB initially developed at CNET, France Telecom, which builds on the ReTINA approach. While it has a modular and extensible architecture, Jonathan is implemented in Java as a classical object-oriented framework without taking advantage of component technologies. In practice, this means that it is difficult to evolve or replace the framework because of implicit and transitive dependencies. Moreover, the binding framework in the latest Jonathan version seems to be mainly geared towards classic first-party bindings. That is, it involves binding a reference in order to obtain an access path to its corresponding object, which may be a remote and/or logical object.

Hector [2] is another distributed processing environment based on RM-ODP that notably supports complex, multi-party bindings. Hector bindings specify roles, defined as placeholders for interfaces, and communication patterns between them, and can be used to describe rich, high-level tasks (e.g., an electronic contract between parties involved in a real-estate purchase). However, Hector does not support any notion of component-orientation. The programming model available to applications follows RM-ODP and does not comply with any commercial standards. Moreover, it is distinct from the programming model available in the support infrastructure, which is implemented in Python and does not have a component architecture. One implication of this is that it is not straightforward to implement complex BTs by recursively building on simpler BTs.

The Regis [6] system can be extended with implementations of various interaction styles. A recent version of Regis defines a language, Midas [22], for specifying interaction styles in terms of asynchronous messages and state machines. The main limitation of both these systems is that they follow the Darwin [6] binding model, which imposes that bindings are always between two types of participants (i.e., a client and a server). Although the model can capture a large class of useful interaction styles (e.g., RMI, message ports, event dissemination), it precludes multi-party, complex binding types, such as group communications or auction protocols.

The DIMMA [8] platform provided an explicit binding model with application-controlled QoS, but the API offered only two kinds of bindings, namely operational and stream bindings. The FlexiNet [12] platform also supports explicit binding but is restricted to first-party, RMI bindings. FlexiBind [11] extends FlexiNet with dynamic binding configuration based on pluggable policies.

Finally, and more recently, the authors in [3] propose building distributed applications using "medium" components, which encapsulate reusable communication services (e.g., video broadcast, voting, mailboxes). Medium components are analogous to BT components in our work, and they are also specified using UML collaborations. Medium components are implemented as sets of role managers that communicate among themselves using an underlying middleware platform. Each participant is associated with one role manager. This model seems to restrict the range of binding models

that can be accommodated (e.g., it cannot accommodate dynamically-established bindings as a result of interface references entering address spaces). Moreover, the underlying middleware for inter-manager communication provides a fixed set of high-level interaction primitives (i.e., calling operations on a set of managers synchronously and asynchronously), which limits the implementation options for medium developers and can impact efficiency.

8. Evaluation and Conclusions

Performance evaluation of an open framework is inherently problematic as the framework can be used to built arbitrary software structures. However, in terms of the efficiency of interactions over bindings implemented with the framework, we are confident that no undue overhead is incurred by the framework itself. Evidence for this was presented in [5] which showed that IIOP invocation in OpenORB (which used an earlier version of the Binding CF) was actually faster than highly respected ORBs such as TAO or Orbacus. Of course most of the credit for this goes to the underlying communications and resources layers, but it does demonstrate that the binding framework is not imposing unacceptable overhead. It is also problematic to evaluate the overhead of binding establishment time in an environment where overhead is very much a function of individual BTs. But again, we can report that iref generation and resolution for a simple RMI binding takes 11ms and 38 ms respectively[2], which is entirely comparable to that of Microsoft DCOM.

In qualitative terms, we have used the framework to construct a representative selection of BTs including an RMI BT, a publish/ subscribe BT, an auction BT, a group communication BT and a message queuing BT. Furthermore, we have investigated, at least to the level of detailed design, a wide range of others as reported in [21]. In all cases so far investigated, the required functionality has been relatively straightforward to accommodate within the constraints imposed by the Binding API and the BT Contract. This gives us confidence that we have indeed produced a generally useful facility. To further test this assertion, we have plans to apply the framework in a number of areas under the auspices of future projects. In particular, we are exploring the use of the framework to define e-Science related collaborations for GRID computing [10], and to support ad-hoc interactions in environments in which clients must intact with services without prior knowledge of the either the service discovery or access protocols used by the services.

We conclude by summarising the major benefits of binding frameworks in general, and of our solution in particular. The main benefit of a well-designed binding framework is that common interaction patterns can be captured as reusable services, thus providing a structured means whereby the level of abstraction of middleware platforms can be raised to meet emerging needs. A good framework should impose just the right amount of structure: enough to guide the design of BTs so that they can be easily understood and take advantage of generic interfaces and services (including building on existing BTs), but not so much that the preconceptions of the framework restrict the scope of future BTs. We believe that our framework broadly achieves these goals. In addition, our framework offers two additional benefits. First, the framework is designed to have minimal and explicit context dependencies and thus it can be reused in multiple middleware architectures. Second, when integrated into the OpenORB architecture, it provides a convenient, flexible and extensible set of support services to simplify the implementation of BTs.

[2] Tests were performed on an Intel Pentium III PC equipped with 256Mb RAM and rated at 999Mhz. The operating system used was Microsoft Windows XP Professional.

References

[1] J. Ø. Aagedal, E. Ecklund, “Modelling QoS: Towards a UML Profile”, *UML 2002*, Dresden, Germany, September 20 - October 4, 2002. Springer LNCS 2460, ISBN 3-540-44254-5, pp. 275-289.

[2] A. Bond, D. Arnold, and M. Chilvers, “Multi-Party Binding in an ODP World”, *Int’l Conf. Open Distributed Processing*, May 27-30 1997, Toronto, Canada.

[3] E. Cariou, A. Beugnard and J.-M. Jézéquel, “An Architecture and a Process for Implementing Distributed Collaborations”, *6th IEEE Int’l Enterprise Distributed Object Computing Conf.* (EDOC 2002), September 17 - 20, 2002, EPFL, Switzerland

[4] M. Clarke, G.S. Blair, G. Coulson, N. Parlavantzas, "An Efficient Component Model for the Construction of Adaptive Middleware", *Proc. IFIP / ACM Int’l Conf. Distributed Systems Platforms (Middleware'2001)*, LNCS 2218, Heidelberg, Germany, November 2001, pp. 160.

[5] G. Coulson, G.S. Blair, M. Clarke, N. Parlavantzas, "The Design of a Highly Configurable and Reconfigurable Middleware Platform", *ACM Distributed Computing Journal*, Vol 15, No 2, April 2002, pp 109-126.

[6] S. Crane. Dynamic Binding for Distributed Systems, PhD Thesis, Imperial College, Univ. of London, March 1997.

[7] F.Dang Tran, B. Dumant, F. Horn, and J.B. Stefani. “Towards an extensible and modular ORB framework”. *Workshop CORBA use and evaluation*, ECOOP'97, Jyväskylä, Finland, June 1997

[8] D. Donaldson, et al.,, "DIMMA - A Multi-Media ORB", *Proc. IFIP Int’l Conf. Distributed Systems Platforms and Open Distributed Processing* (Middleware '98), The Lake

District, UK, Springer-Verlag, 15-18 September 1998, pp. 141-156.

[9] B. Dumant, F. Dang Tran, F. Horn, and J.B. Stefani, "Jonathan: an open distributed processing environment in Java", *Proc. IFIP Int'l Conf. Distributed Systems Platforms and Open Distributed Processing* (Middleware '98), Springer-Verlag, The Lake District, U.K., September 1998, pp 175-190.

[10] I. Foster, C. Kesselman, S. Tuecke, "The Anatomy of the Grid: Enabling Virtual Organizations, *Int'l Journal of Supercomputer Applications*, Vol 15, No 3, 2001.

[11] Ø. Hanssen and F. Eliassen, "A Framework for Policy Bindings", *Proc. DOA'99*, IEEE Press, Edinburgh, September 1999.

[12] R. Hayton, A. Herbert and D. Donaldson, "Flexinet: a flexible, component oriented middleware system", *Proc. 8th ACM SIGOPS European Workshop: Support for Composing Distributed Applications*, Sintra, Portugal, 7-10 September 1998.

[13] ISO, ITU, *Open Distributed Processing – Reference Model, ISO/IEC 10746-1, 2, 3, 4 | ITU-T Recommendation X.901, X.902, X.903, X.904*, 1995-96.

[14] Microsoft, COM Home Page, http://www.microsoft.com/com/default.asp (current June 2003).

[15] Object Management Group, *CORBA Component Model, OMG Document formal/2002-06-65*.

[16] Object Management Group, *CORBA Event Service v1.0, OMG Document formal/2000-06-15*.

[17] Object Management Group, *Audio/Video Streams, v1.0, OMG Document formal/2000-01-03*.

[18] Object Management Group, *Unified Modeling Language (UML), Version 1.4, OMG Document formal/2001-09-67*.

[19] N. Parlavantzas, G. Coulson and G.S. Blair, "An approach to building reflective component-based middleware platforms", *Microsoft Summer Research Workshop*, Cambridge, U.K., September 9-11, 2002.

[20] N. Parlavantzas, G. Coulson and G.S. Blair, "A Resource Adaptation Framework For Reflective Middleware", *Proc. 2nd Intl. Workshop Reflective and Adaptive Middleware* (located with ACM/IFIP/USENIX Middleware 2003), Rio de Janeiro, Brazil, June, 2003

[21] N. Parlavantzas, *An extensible binding framework for component-based middleware*, tech report, Distributed Multimedia Group, Lancaster University, 2002.

[22] N. Pryce and S. Crane. "Component Interaction in Distributed Systems". *IEEE Fourth Int'l Conf. on Configurable Distributed Systems*, Annapolis, Maryland, USA, May 1998, pages 71-78.

[23] Sun Microsystems, Java Message Service API, http://java.sun.com/products/jms/ (current June 2003).

[24] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley, 1998.

[25] P. Wurman, M. Wellman, et al., "A Control Architecture for Flexible Internet Auction Servers," *First IAC Workshop Internet-Based Negotiation Technologies*, Yorktown Heights, New York, 1999.