# IKE 2—Implementing the Stateful Distributed Object Paradigm

J. Russell Noseworthy
Object Sciences Corporation
j.russell.noseworthy@objectsciences.com

***Abstract*—**
**This paper describes IKE 2, which is distributed object computing middleware that supports the creation of interoperable real-time distributed applications. These applications appear in many domains, such as telecom, aerospace, military testing and training ranges, and financial services. IKE 2 combines the concepts of CORBA distributed objects and anonymous publish-subscribe data dissemination to provide a programming abstraction known as a *stateful distributed object* (SDO).**

**Every SDO can have both a *remote method interface* and *publication state*. The remote method interface allows client applications to invoke methods on target objects efficiently without concern for their location, programming language, OS platform, communication protocols and interconnects, or hardware. The publication state of a given SDO is disseminated to applications that have expressed their interest by subscribing to certain characteristics, such as the type of the SDO. Subscribers can read the publication state of an SDO as if it were a local object. The SDOs provided by the IKE 2 metaobject model support inheritance from other SDOs, containment of other SDOs, and references to other SDOs.**

**IKE 2 is implemented in C++. The API relies heavily on compile-time type-safety to help ensure reliable behavior at run-time—a critical feature of any real-time system. Automatic code generation is used to provide the high-level abstractions without unduly burdening application programmers.**

**Keywords: Distributed Object, Middleware, Distributed Object Middleware, Publish-Subscribe Middleware, SDO, Stateful Distributed Object, CORBA, Event Service**

## I. Introduction

### A. Distributed Communication Models

Distributed object computing (DOC) has been a popular paradigm for developing complex distributed systems for many years, as witnessed by the success of Java RMI (28) and CORBA (20). Distributed objects combine powerful object-oriented design and programming abstraction and decomposition concepts with the illusion of location transparency. This combination yields objects whose interfaces expose methods that can be invoked from any process in the distributed system in a manner identical[1] to how methods are invoked on objects that are local to the process. Figure 1 illustrates the conventional DOC two-way synchronous communication model.

[1]It could be argued that the possibility of a run-time exception arising from the invocation of a method on a distributed object due to the failure of some component of the distributed system constitutes a marked difference between programming with a distributed object and programming with a purely local object. However, in general, even local method invocations can raise exceptions, so in that sense, there is no difference.

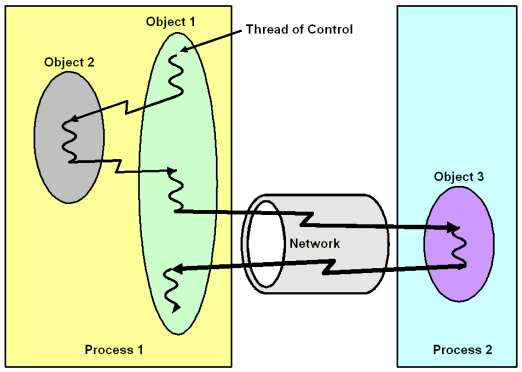


Fig. 1. Conventional DOC Communication Model.

While the DOC paradigm is a powerful tool to address the challenges of developing distributed systems, it is not the only approach. In particular, many data-centric distributed systems employ a technique known as *anonymous publish and subscribe data exchange*, or, more tersely, *publish-subscribe*. Publish-subscribe systems consist of the following components:

- **Publishers** are event sources, *i.e.*, they generate the events that are propagated through the system. Publishers may need to describe the specific type of events they generate *a priori* or they can generate generic events.
- **Subscribers** are the event sinks of the system, *i.e.*, they receive the events sent by publishers. Some architecture implementations require subscribers to declare filtering information to designate the events they are interested in receiving.
- **Event channels** are components in the system that propagate events from publishers to subscribers. In distributed systems, event channels can propagate events across distribution domains to remote subscribers. Event channels can perform event filtering, correlation, routing, QoS enforcement, and fault management.

Figure 2 illustrates the relationships and information flow between these three components.

In *anonymous* publish-subscribe systems, publishers are unaware of which, if any, subscribers are receiving the data they produce. Likewise, subscribers are unaware of which, if any, publishers produce the data they desire. It is also possible to construct publish-subscribe systems that are not anonymous. Providing that additional information comes at a non-trivial ex-

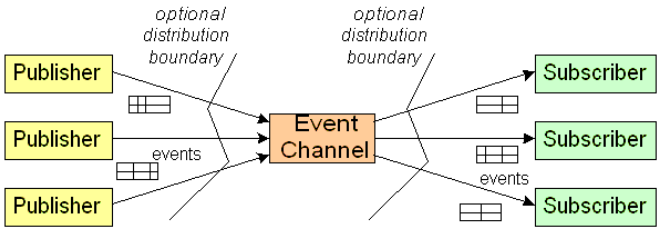


Fig. 2. Relationships Between Components in a Publish-Subscribe Model

pense, however, in terms of system performance. Moreover, it only adds marginal utility.

Publish-subscribe systems can disseminate data in many ways, ranging from an individual publisher to an individual subscriber to multiple publishers to multiple subscribers. Data dissemination is typically accomplished in a message- or event-based fashion, meaning that the data arrives asynchronously and is ephemeral, *i.e.*, if users do not take it upon themselves to store the data then it is lost since it is not stored persistently by the event channel. Examples of publish-subscribe systems include NDDS (22), the CORBA Event Service (19; 21; 10), the CORBA Notification Service (17; 9; 26), Sienna (4), and the HLA RTI-NG (11).

In the standard CORBA Event Service, subscribers cannot specify the type of data they are interested in receiving. In most publish-subscribe systems, however, including the CORBA Notification Service, subscribers can specify which types of events they want to receive. This filtering capability provides a richer programming paradigm to users, though it increases the implementation complexity of the publish-subscribe system.

### *B. Combining the Distributed Object and Publish-Subscribe Programming Paradigms*

DOC systems and publish-subscribe systems both provide users with powerful programming abstractions. Yet each provides something the other lacks. Traditional remote procedure call (RPC) toolkits, such as Sun RPC (27), offer no direct support for disseminating data from a single source to multiple destinations. Conversely, traditional publish-subscribe systems, such as NDDS, do not provide the abstraction of objects with a set of methods in their interface. Without direct support for the powerful abstractions provided by object-oriented programming, the difficulty of modeling complex concepts and designing sophisticated systems is increased.

Combining the distributed object and publish-subscribe programming paradigms yields a new paradigm that provides the strengths of each. At the same time, the new paradigm frees the programmer from the common chore of explicitly storing the data that arrives as part of the publish-subscribe system. This new programming paradigm introduces the concept of a *stateful distributed object*, or SDO, which is shown in Figure 3.

As shown in Figure 3, an SDO has two parts:

- **An object interface** that provides a location-transparent interface to its methods. This interface is consistent with the capabilities shown in Figure 1.
- **Publication state**, which is data that is disseminated from the creator of an instance of an SDO to all parties that have indicated their interest in that SDO's data through a subscription. This capability of an SDO provides the capabilities shown in Figure 2.

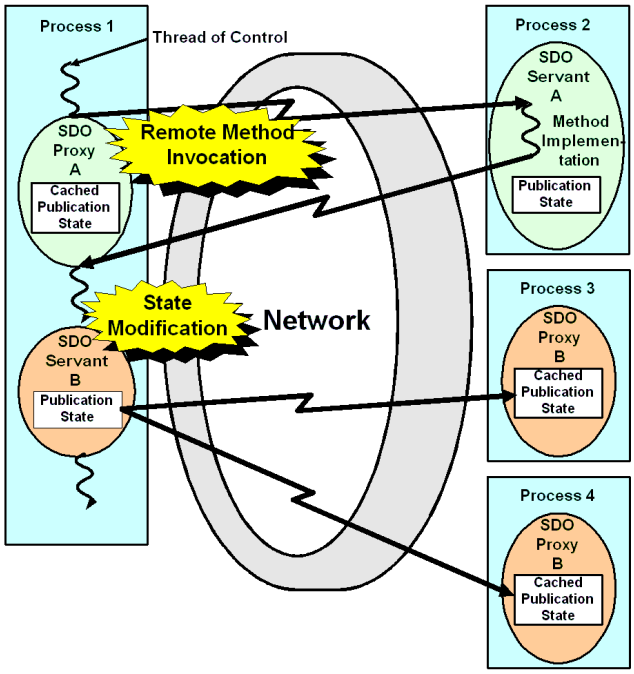


Fig. 3. Fundamental Aspects of an SDO

Interested subscribers can receive references to an SDO, which they can use to invoke methods on its interface, as if it were a reference to a CORBA distributed object. In addition, an SDO reference provides programmers with the ability read the publication state of the SDO as if it were local data, as can be done in many distributed shared memory systems (15; 1). Thus, an SDO is an object that are easily modeled in UML, even though both the methods and attributes (publication state) of an SDO are location independent. As modifications to a given SDO's publication state are disseminated from its publisher, subscribers are notified that new values of the publication state data are available.

The abstract concept of the SDO has been implemented in a middleware system known as IKE 2, which is the second middleware prototype of the Test and Training Range ENabling Architecture (TENA) (6), an architecture being developed by the Foundation Initiative 2010 (FI 2010) project (6). FI 2010 is an interoperability initiative of the Director, Operational Test and Evaluation (DOT&E), funded through the Central Test and Evaluation Investment Program (CTEIP). The FI 2010 vision is to enable interoperability among ranges, facilities, and simulations in a quick and cost-efficient manner, and to foster reuse of range assets and of future range system developments. The

purpose of IKE 2 is to support the creation of interoperable real-time object-oriented distributed system applications, such as those needed by military testing and training range systems.

The remainder of this paper is organized as follows: Section II describes the SDO concepts that underlie the IKE 2 metaobject model and the TENA definition language; Section III describe the fundamental concepts used in IKE 2; Section IV explores key aspects of IKE 2's implementation; Section V outlines future work on IKE 2; and Section VI presents concluding remarks.

## II. Overview of IKE 2 and TENA

Since IKE 2 provides an object-oriented programming paradigm, its metaobject model is similar to the metaobject model of CORBA, C++, and Java. As a DOC framework, IKE 2 requires an interface definition language (IDL), which is similar to CORBA IDL. These characteristics of IKE 2 are explained below.

### A. The IKE 2 Metaobject Model

Metaobject models describe the concepts and features that are supported by a particular object system. For example, CORBA has a metaobject model that describes how CORBA interfaces may inherit from each other, have location-transparent references, and support introspection.[2] The IKE 2 metaobject model extends the CORBA metaobject model by distinguishing between a pure interface and an SDO. In the IKE 2 metaobject model, an interface has the same general properties of a CORBA interface, *i.e.*, it is a declaration of a collection of methods. However, an SDO has publication state as well as methods. Single inheritance between SDOs is supported. In addition, an SDO may multiply inherit from any number of interfaces. This is analogous to the `extends` and `implements` features of the Java metaobject model.

The publication state of an SDO may be composed of fundamental data types, as well as complex data types (*e.g.*, structures). Moreover, an SDO may *contain* other SDOs. The IKE 2 metaobject model therefore supports the UML concept of *composition* (3).[3] The publication state of an SDO is always updated atomically, including the publication state of any SDOs that are contained in the SDO whose publication state is being modified. By ensuring the atomicity of publication state updates, IKE 2 subscribers are prevented from reading invalid publication state, *e.g.*, publication state composed of partly old and partly new data.

In addition to composition of SDOs, the IKE 2 metaobject model provides the ability for an SDO to contain a reference to another SDO. The semantics of this association between SDOs is referred to in UML as *aggregation* (3) and in the C++ metaobject model as a pointer. In C++, an object may contain a pointer to another object. The pointer may be changed to refer to a different object. Likewise, the pointer may sometimes be null. In IKE 2, a reference contained in an SDO may be changed to refer to an different SDO or even a special null SDO. Since the referent SDO is only "pointed at" by the referring SDO, the referent SDO's publication state is not updated atomically with the referring SDO when state changes occur.

IKE 2's metaobject model supports containment of fixed-length arrays of fundamental and complex data types, as well as SDOs. It also supports containment of variable-length sequences of fundamental and complex data types, as well as of references to SDOs. Containment of variable-length SDOs is not yet supported. Likewise, support for introspection, dynamic invocation of methods (as with the CORBA dynamic invocation interface), and self-describing arbitrary types (*e.g.*, the CORBA Any) are envisioned as part of the IKE 2 metaobject model, though they are not yet implemented.

### B. TENA Definition Language

An interface definition language (IDL) provides a strongly typed format for declaring interfaces, methods, types, constants, and other information exchanged between clients and servers. An IDL is not a full-fledged programming language and hence has no means to specify detailed implementation logic, such as branching or control flow. This ensures that a fundamental tenet of object-oriented programming is observed: the separation of interface from implementation. An IDL compiler also automates the generation and optimization of low-level marshaling and demarshaling code, which can be tedious and error-prone to write and debug manually (8).

Programmers of CORBA systems provide specific declarations of CORBA objects using CORBA IDL. In much the same way, IKE 2 programmers express specific declarations of SDOs (and interfaces) using the TENA definition language (TDL). As with the IKE 2 metaobject model, TDL extends CORBA IDL with new capabilities, such as `class`, `extends`, and `implements`. Moreover, TDL uses the `attribute` keyword to indicate elements of an SDO's publication state[4]. Note that, as is the case with CORBA IDL, TDL is not a complete programming language. It is instead a language used to declare SDOs.

Figure 4 on the following page shows an example of TDL. In Figure 4, `Participant` implements the `Controllable` interface; `Sensor` inherits from `Participant`; and `Platform` inherits from `Participant` and contains a `Sensor`. `Participant` has no methods (beyond the method implemented from the `Controllable` interface), while `Sensor` and `Platform` each have one additional method.

TDL is not the first extension to CORBA IDL designed to address issues of object state. The CORBA Persistent State Service (18) introduced the so-called *Persistent State Description*

[2] Since describing the complete CORBA metaobject model is beyond the scope of this paper, readers are referred to (20).

[3] At present, IKE 2 does not allows parts with non-fixed multiplicity, which is allowable in UML.

[4] Since the elements of the publication state can be inferred without the `attribute` keyword, it may be removed in future versions of TDL.

```
module Example
{
    interface Controllable
    { string initialize(); };
    class Participant
        : implements Controllable
    {
        attribute string name;
        attribute long ID;
    };
    class Sensor : extends Participant
    {
        attribute string state;
        string point(in double azimuth);
    };
    class Platform : extends Participant
    {
        attribute double fuel;
        attribute Sensor longRangeSensor;
        void move(in double x);
    };
};
```

Fig. 4. TDL example showing an interface and three SDOs.

*Language* (PSDL). While TDL and PSDL are both concerned with object state, they differ fundamentally in the aspects of object state they address. PSDL targets the server-side storage of object state, while TDL describes objects whose active state is disseminated to clients.

## III. Fundamental Concepts in IKE 2

IKE 2 weaves together many concepts to form a coherent system. This section describes the most important of those concepts, namely the SDO servant, the SDO proxy, the user-specifiable client-side callbacks, the publication state consistency policies, and the concepts of executions and sessions.

### A. SDO Servants and Proxies in IKE 2

IKE 2 defines servant and proxy constructs for SDO that are analogous to the servant and proxy in CORBA. Below, the purpose of these constructs is explained.

*1) SDO Servants in IKE 2 :* Servants in IKE 2 are the objects in the target programming language, *i.e.*, C++, that implement SDOs. As is the case with a CORBA servant, an SDO servant requires servant programmers to provide implementations of the methods on the SDO. However, SDO servants differ from CORBA servants in two important ways: SDO servants are created using a factory and SDO servants have publication state that can be modified. These differences are further explained below.

*a) Creating SDO servants with a factory.:* A CORBA servant can be created in many ways, *e.g.*, on the stack, on the heap, by a servant locator interceptor, etc. In contrast, an SDO servant is always created via a *factory* (7). This design ensures that the publication state provided by the IKE 2 middleware system and the implementation of the servant's methods provided by the programmer can be combined properly to form a complete SDO servant.

The structure imposed by using a factory also ensures that the implementation of the SDO servant's methods can access the SDO servant's publication state. To understand why this is important, consider the `Platform` SDO shown in Figure 4. It is reasonable to expect that most implementations of the `move()` method on a `Platform` would want to modify the value of the `Platform`'s fuel. The implementation of the `Platform`'s methods therefore requires access to the `Platform`'s publication state.

*b) Modification of an SDO servant's publication state.:* As mentioned in Section I-B, the publication state of an SDO servant is updated atomically. This concept is quite common in distributed systems and concurrent systems. Without native support for atomic modification of multiple publication state attributes, it would be possible for IKE 2 applications to observe inconsistent values of publication state that mixed old and new values. Note that this atomicity must extend throughout any chain of contained SDOs.

In addition to the atomicity of multi-attribute modifications, there is another prominent characteristic of an SDO servant's publication state: only the SDO servant is capable of modifying the publication state of the SDO.

*2) SDO Proxies in IKE 2 :* Proxies in IKE 2 are the objects in the target programming language that provide access to remote SDO servants as if they were local objects. The IKE 2 middleware system implements SDO proxies for use by the IKE 2 programmer. An SDO proxy provides location-transparent access to both the corresponding SDO servant's methods and publication state.

As discussed above, only an SDO servant can modify the publication state of the SDO. An SDO proxy may *read* the SDO's publication state, but may not *modify* the SDO's publication state. This constraint is not a significant limitation in practice since an SDO proxy *can* invoke an SDO's methods. If it were important that proxies to certain SDOs be able to modify the SDO's publication state in a given IKE 2 application, a mutator method could be provided on the particular SDO.

### B. Asynchronous Notification of SDO Events

Throughout the lifetime of an SDO, there are several noteworthy events that are of potential interest to it's subscribers. These events include notification of the creation and destruction of the SDO, notification that the SDO no longer meets (or once again meets) the interest criteria specified by the subscriber, and notification that the publication state of the SDO has changed. These various notifications are discussed below.

- **SDO Discovery Notification**—Instances of SDO servants are *discovered* in response to a subscription, in keeping with the publish-subscribe paradigm. The IKE 2 programmer declares their interest in particular SDO servants to the IKE 2 middleware. The IKE 2 middleware in turn provides individual notification to the IKE 2 programmer of every SDO servant instance that matches their interest. When a new SDO servant that matches a subscription is instantiated in the system, the IKE 2 middleware generates a new discovery notification. Each discovery notification provides a new SDO proxy to the IKE 2 programmer.
- **SDO Scope Notifications**—Throughout the life of the application, it is possible that the a given instance of an SDO will cease to match the IKE 2 programmer's interest, either due to some change in the SDO or due to a change in the programmer's interest. When this occurs, the IKE 2 middleware provides notification to the programmer that the particular SDO is *out of scope*. If that same SDO subsequently matches the programmer's interest once again, then the IKE 2 middleware delivers notification of this fact via an *in scope* event.
- **SDO Destruction Notification**—In addition to being instantiated, SDOs can also be destroyed. When this occurs, the IKE 2 middleware generates a destruction notification event. Note that destruction notification events pertaining to SDOs that are not currently of interest to the IKE 2 programmer (*i.e.*, are out of scope) are not generated.
- **SDO Publication State Change Notification**—The SDO programming paradigm is closely aligned with the concept of distributed shared memory. Unlike classic message- or event-based systems which deliver notification whenever new data arrives, a classic distributed shared memory system provides no notification when data changes. Indicating when an SDO's publication state has been modified is so useful, however, that the IKE 2 middleware provides the programmer precisely this information for every SDO presently in scope. This eliminate the necessity to *poll* the publication state to determine if it had been modified—a practice that is hard to program efficiently, scalably, and correctly.

At present, subscription interests in IKE 2 are declared by SDO type. For example, consider the TDL shown in Figure 4. IKE 2 programmers could declare their interest in SDOs of type `Participant`. The IKE 2 middleware system would then generate discovery notifications for each instance of an SDO servant of type `Participant` presently in the system, as well as discovery notifications for any future `Participant`s that are instantiated.

Moreover, the IKE 2 middleware would generate discovery notifications for each instance of an SDO of type `Sensor` and type `Platform`—because `Sensor` and `Platform` inherit from `Participant`. The SDO proxies provided to the IKE 2 programmer in these cases would be of type `Participant`, however, because that is what the IKE 2 programmer requested by subscribing to `Participant`s. If other IKE 2 programmers were to subscribe to `Sensor`s, they would receive discovery notifications for each SDO servant instance of type `Sensor` in the system, including the `Sensor`s contained in any SDO servant instances of type `Platform`.

Interests in IKE 2 are presently type-based, and it is not possible for an SDO to change its type. Thus, scope notifications would currently only be generated as the result of unsubscribing to a type. Due to their limited utility at this time, scope notifications are not implemented in this version of the IKE 2 middleware.

In addition to the ability to discover SDOs based on their type, IKE 2 provides the ability to obtain SDO proxies by name, using the IKE 2 SDO Naming Service. The IKE 2 SDO Naming Service is analogous to the CORBA Naming Service. An IKE 2 user has the ability to register an SDO servant in the IKE 2 SDO Naming Service with some name. In so doing, other IKE 2 users can then obtain an SDO proxy to that SDO servant using the name registered in the IKE 2 SDO Naming Service.

User applications are often more easily constructed if they have the ability to control when the notification of asynchronous events are delivered. To provide flexibility to the IKE 2 users, IKE 2 uses the Half-Sync/Half-Async pattern (25) to queue all asynchronous event notifications, delivering them only when expressly requested by users.

### C. SDO Publication State Consistency Policies

Whenever data (*i.e.*, state) is replicated, the question of the type and degree of data consistency must be addressed. This question has been considered in the context of multi-processor caching systems, in the context of distributed database systems, and in the context of distributed shared memory systems. In all cases, decisions about the data consistency guarantees made by the system to the users of that system must be made. IKE 2 is no exception, so its middleware is therefore designed to support the following state consistency policies:

- **Cached policy**—In this policy, each SDO proxy maintains a local cache of the SDO's publication state. To reduce the latency in reading an SDO's publication state, IKE 2 transmits the publication state of an SDO from the servant (where it is written) to each of the proxies (where it can only be read). The proxies cache that publication state, resulting in multiple copies of an SDO's publication state throughout the distributed system. If a version of the publication state older than the version presently cached was received, the older version would simply be ignored in favor of the newer version. This easy-to-understand implementation has been used for years in hardware systems, where it is referred to as PRAM, for *pipelined* RAM (13; 14).[5]

[5] PRAM systems do not restrict the ability to modify data to a single source, thereby resulting in an even greater probability that the various caches do not hold the identical value for any given moment in time.

Since the communication time in a distributed system is not instantaneous and generally non-uniform, at any given moment in time the various caches likely do not all hold a consistent value. In particular, it is quite possible that *none* of the caches hold the same value as the "master copy" held by the SDO servant.

- **Queued policy**—This policy is similar to the cached policy, except that "late arriving" publication state updates are not ignored. Instead they are appended to a queue of publication state updates. For certain applications, such as data logging, a "late arriving" publication state (*i.e.*, a version of publication state older than the cached version currently held) is still valuable and should not be ignored.
  When the queued policy is enabled, each read of an SDO's publication state by an SDO proxy consumes an element from the head of this queue. The only time a read does *not* consume the head of the queue is when there are no other versions of the publication state presently in the queue. If a new version is received, the previously read version at the head of the queue would be discarded. In this policy, an application that repeatedly read the publication state of an SDO would read—at least once—every version of the publication state written by the SDO servant, although not necessarily in the order written by the SDO servant.[6]

### D. IKE 2 Executions and Sessions

Executions in IKE 2 are a grouping of participants with some collection of common resources, such as the IP multicast groups assigned to the particular execution. The unit of participation in an execution is referred to as a *session*, which are local to a single process. A session provides the context for publishing and subscribing to SDOs. Moreover, sessions are the logical place to explore future functionality, such as security credentials, user-specified threading policies, and user-specified communication characteristics.

A single LAN can simultaneously support multiple executions. A single process can simultaneously support multiple sessions, possibly from distinct executions. Figure 5 depicts the relationships between executions, sessions, processes, and hosts processors. Sessions in a single process may belong to multiple Executions. In Figure 5, four hosts with one or two processes are depicted. Each process has from one to three sessions. Each process participates in from one to three distinct executions.

## IV. Key Points of the IKE 2 Implementation

The SDO concepts described in Section III paper have been implemented in a DOC middleware framework called IKE 2. This section describes key points of the IKE 2 implementation.

[6] In actuality, if the publication state updates were being delivered using an unreliable transport mechanism, it is possible that some publication state updates could be lost, and hence *not* read by the SDO proxy.

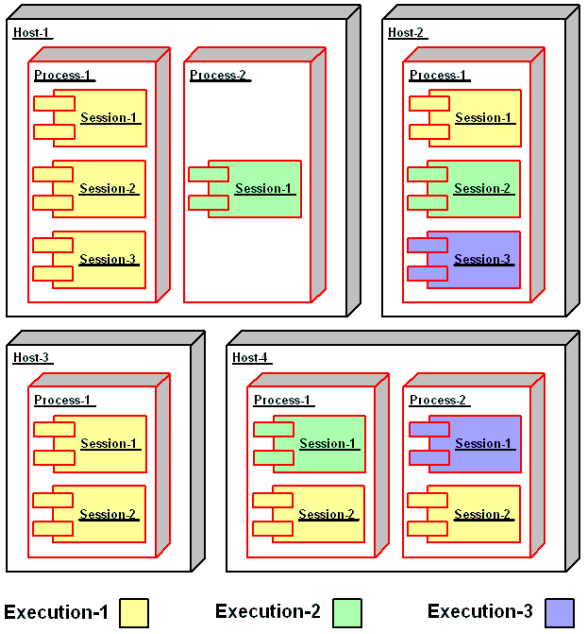


Fig. 5. IKE 2 supports multiple Sessions in a process.

### A. Compile-time Type-safety

The IKE 2 middleware API relies heavily on compile-time type-safety to help ensure reliable behavior at run-time. Reliability is a critical feature of most systems, but is especially critical in a real-time system, such as many of those used at military testing and training ranges. Strong compile-time type-safety eliminates an entire class of run-time errors that can occur due to dynamic type mismatch (*e.g.*, `bad_cast` exceptions).

### B. DOC Middleware Infrastructure

Implemented entirely in standard C++, IKE 2 achieves cross-platform portability over a variety of UNIX and Windows platforms using ACE+TAO (23; 5), which are freely-available open-source DOC middleware. ACE is a highly-portable collection of C++ classes and frameworks that implement many useful patterns for the programming of distributed systems (25). TAO is a popular real-time CORBA ORB that is implemented using the patterns and frameworks provided by ACE. Figure 6 illustrates the relationship between IKE 2 and ACE+TAO.

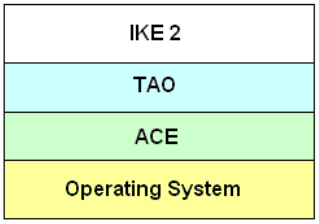


Fig. 6. Layers in the IKE 2 Middleware Architecture.

Given the strong grounding the SDO concept has in the CORBA DOC middleware paradigm, using a CORBA ORB as

the foundation for the IKE 2 middleware is an obvious choice. IKE 2 takes advantage of the following features of TAO:

- IKE 2 uses CORBA valuetypes (16) to implement SDO publication state.
- IKE 2 uses TAO's real-time CORBA event service (21) as its data dissemination engine.
- IKE 2 uses the TAO Naming Service to allow users to obtain SDO proxies by name.
- IKE 2 uses TAO's buffered oneways to reduce blocking during data writes in those cases where the readers are heavily loaded.
- IKE 2 uses TAO's optimizations for efficient copying of sequences of octets to reduce memory copy overhead.

### C. Automatic Code Generation

IKE 2 makes extensive use of automatic code generation. This capability provides the high-level abstractions of the SDO without unduly burdening application programmers who use IKE 2. Just as CORBA automatically generates code using an IDL file as input to an IDL compiler, IKE 2 automatically generates code using a TDL file as input to the TDL compiler. Automatic code generation makes the IKE 2 middleware practical for use. Without the automatic code generation, implementing an SDO would be tedious and error-prone.

For each SDO declared in a TDL file, the IKE 2 TDL compiler generates several dozen C++ classes as well as an IDL file. Figure 7 shows an example of the automatically generated IDL for the `Participant` SDO shown in the TDL example of Figure 4. In this figure, the CORBA interface used to provide the SDO's methods and the CORBA `valuetype` used to implement the SDO's publication state are clearly evident.

```
#include <Participant/Participant.idl>
#include <Sensor/Sensor.idl>
module Example
{
  module PlatformIDL
  {
    valuetype PublicationState
      : ParticipantIDL::PublicationState
    {
      public double fuel;
      public SensorIDL::PublicationState
        longRangeSensor;
    };
    interface Servant : Participant::Servant
    {
      void move(in double x);
    };
  };
};
```

Fig. 7. Example of the IDL automatically generated from TDL.

### D. SDO Publication State Dissemination

The means by which the IKE 2 middleware disseminates publication state from publishers to interested subscribers is an implementation detail that is (theoretically) not visible to IKE 2 users. What must be guaranteed, however, is that updates to multiple attributes of an SDO's publication state occur *atomically*. One way to achieve this is to transmit every attribute of the SDO's publication state (including any contained SDOs) any time a modification to one or more of the attributes is committed. This technique is used by the existing IKE 2 middleware. Section V describes more sophisticated approaches.

The actual dissemination of the SDO publication state is performed using the *Distributed Interest-based Message Exchange* (DIME). DIME is an extensible anonymous publish-subscribe message-passing framework. It decouples (1) the manner in which interests in a publish-subscribe system are used to describe the desired data from (2) the means by which the desired data is routed to its destinations.

As is the case with IKE 2, DIME is implemented in C++ using ACE+TAO as its foundation. To facilitate its publish-subscribe data dissemination, DIME makes extensive use of TAO's real-time CORBA event channel (21). Presently, DIME supports distribution schemes based on reliable TCP/IP and unreliable UDP multicast/broadcast.

## V. Future Work on IKE 2

Future work planned for IKE 2 spans a broad range of topics. Issues involving *quality of service* (QoS) and other communication characteristics are being designed. User-customized multi-threading support will be improved, as will IKE 2's use of real-time CORBA (24). The features of real-time CORBA will provide the IKE 2 user with finer-grained control over the numbers and priorities of threads that handle the SDO servants than is possible in the current versions. Likewise, the envisioned QoS features will allows IKE 2 users to control the manner in which data is delivered, adjusting characteristics such as latency, bandwidth, and message delivery ordering.

Future versions of IKE 2 will also support a Java language mapping, introspection, dynamic method invocation, a generalized interceptor framework, a generic data logging mechanism, support for in and out of scope notifications, the addition of data streams and messages, better diagnostic support, and basic security features. Moreover, the addition of complex, user-definable interests is planned to allow users to subscribe to SDOs based not solely on type, but on other characteristics as well, such as the ability to subscribe to all the `Platform`s within half a mile of a particular position.

A reliable multicast protocol suitable to disseminate publication state data is planned. In addition, the IKE 2 development team is designing sophisticated implementations of publication state update data routing techniques, such as protocols that transmit only update deltas, *i.e.*, transmit only those attributes of the publication state that actually were modified. In

general, that problem is quite challenging due to issues of message loss and out-of-order message arrival. Moreover, the containment of SDOs by other SDOs, as well as the inheritance of SDOs, complicates the decisions of where data should be sent. In turn, this can result in the need to transmit multiple small messages, as opposed to the one "large" message that is transmitted in non-delta-based schemes.

Finally, the need for advanced SDO publication state consistency policies is under consideration. IKE 2 does not presently provide the ability to ensure any form of data consistency between multiple SDO instances. Consider two simple SDOs, A and B, where B ideally is an exact duplicate of A at all times. Due to any number of reasons, such as network delay or processor load, even a single host in an IKE 2 execution that subscribes to both A and B cannot be guaranteed to have the same values of A and B at the same time. Adding additional hosts that subscribe to A and B only exacerbates the problem.

It may be necessary to implement a stronger form of publication state consistency in IKE 2, such as *strict causal* consistency (15) or *sequential* consistency (12), and perhaps some from of distributed transaction mechanism, such as virtual synchrony and process groups (2).

## VI. Concluding Remarks

Many real-time distributed applications can benefit from an enhanced object-oriented communication model that provides a location-transparent interface to object methods *and* the notion of publication state. The IKE 2 middleware combines the distributed object and publish-subscribe programming paradigms to create a new *stateful distributed object* (SDO) model that provides the strengths of each. The IKE 2 SDO implementation described in this paper augments the standard CORBA metaobject model and interface definition language (IDL) to provide an object possessing both location transparent methods and location transparent *publication state* attributes. The publication state of an SDO is disseminated to interested subscribers who can observe changes an SDO's publication state using callbacks.

Beta testing of early IKE 2 releases is being conducted at numerous military testing and training ranges and initial feedback has been quite positive. Work for subsequent releases is underway. These releases will address such issues as QoS, multithreading, and the use of real-time CORBA (24).

Experience to date indicates that the SDO programming paradigm is useful to not just the military testing and training range community, but to a broad spectrum of other communities as well. As a government-sponsored program, IKE 2 is not encumbered by forces that tend to keep many commercial projects closed. Continued success in the range community will hopefully increase the visibility of IKE 2 and lead to its broad adoption.


## Acknowledgments

This work is sponsored by the Foundation Initiative 2010 Project Management Office (6). This paper describes the work of a team of software professionals, including the author and the following members of original IKE 2 development team: Steve Bachinsky, Jon Franklin, Rob Head, Serge Kolgan, Mike Mazurek, Sean Paus, Ed Powell, and Ed Skees, as well as two new team members: Marvin Greenberg and Rodney Smith. The entire IKE 2 development team would like to thank the FI 2010 project team, in particular George Rumford, Jason Lucas, Kurt Lessmann, Larry Rothstein, and Kevin Alix. Thanks also to Douglas C. Schmidt, who provided extensive comments on earlier drafts of this paper.


## References


[1] H. E. Bal and M. F. Kaashoek. Object Distribution in Orca using Compile-Time and Run-Time Techniques. In *Proc. of the 8th Annual Conf. on Object Oriented Programming Systems, Languages, and Applications*, pp. 162–177, Washington, D.C., Sep. 1993.

[2] K. Birman and R. van Renesse. *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, Los Alamitos, 1994.

[3] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modelling Language User Guide*. Addison Wesley Longman, Inc., Reading, MA, 1999.

[4] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and Evaluation of a Wide-Area Event Notification Service. *ACM Transactions on Computer Systems*, 19(3):332–383, Aug. 2001.

[5] Center for Distributed Object Computing. TAO: A High-performance, Real-time Object Request Broker (ORB). http://www.cs.wustl.edu/∼schmidt/TAO.html, Washington University.

[6] FI 2010. Foundation Initiative 2010. http://fi2010.jcs.mil/.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1995.

[8] A. Gokhale and D. C. Schmidt. Optimizing a CORBA IIOP Protocol Engine for Minimal Footprint Multimedia Systems. *Journal on Selected Areas in Communications special issue on Service Enabling Platforms for Networked Multimedia Systems*, 17(9), Sep. 1999.

[9] P. Gore, R. K. Cytron, D. C. Schmidt, and C. O'Ryan. Designing and Optimizing a Scalable CORBA Notification Service. In *Proceedings of the Workshop on Optimization of Middleware and Distributed Systems*, pp. 196–204, Snowbird, Utah, June 2001. ACM SIGPLAN.

[10] K. S. Ho and H. V. Leong. An Extended CORBA Event Service with Support for Load Balancing and Fault-Tolerance. In *Proceedings of the International Symposium on Distributed Objects and Applications (DOA'99)*, Antwerp, Belgium, Sep. 2000. OMG.

[11] F. Kuhl, R. Weatherly, and J. Dahmann. *Creating Computer Simulation Systems*. Prentice Hall PTR, Upper Saddle River, New Jersey, 1999.

[12] L. Lamport. How to Make a Multiprocessor that Correctly Executes Multiprocess Programs. *IEEE Trans. on Computers*, C-28(9):690–691, Sep. 1979.

[13] R. J. Lipton and J. S. Sandberg. PRAM: A Scalable Shared Memory. Department of Computer Science Technical Report CS-TR-180-88, Princeton University, NJ, Sep. 1988.

[14] R. J. Lipton and D. N. Serpanos. Uniform-Cost Communication in Scalable Multiprocessors. In *Proc. of the Int. Conf. on Parallel Processing*, vol. I, pp. 429–432, Aug. 1990.

[15] J. R. Noseworthy. *A Novel and Efficient Distributed Shared Memory Algorithm for Strict Causal Consistency*. ECSE Dept. Ph.D. thesis, Rensselaer Poly. Inst., Troy, NY, Sep. 1996.

[16] Object Management Group. *Objects-by-Value*, OMG Document orbos/98-01-18 edn., January 1998.

[17] Object Management Group. *Notification Service Specification*, OMG Document telecom/99-07-01 edn., July 1999.

[18] Object Management Group. *Persistent State Service 2.0 Specification*, OMG Document orbos/00-12-07 edn., Jan. 2000.

[19] Object Management Group. *Event Service Specification Version 1.1*, OMG Document formal/01-03-01 edn., March 2001.

[20] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 2.5 edn., Sep. 2001.

[21] C. O'Ryan, D. C. Schmidt, and J. R. Noseworthy. Patterns and Performance of a CORBA Event Service for Large-scale Distributed Interactive Simulations. *International Journal of Computer Systems Science and Engineering*, 2001.

[22] Real-Time Innovations. NDDS: The Real-Time Publish-Subscribe Middleware. http://www.rti.com/products/ndds/ndwp0899.pdf, 1999.

[23] D. C. Schmidt and S. D. Huston. *C++ Network Programming, Volume 1: Mastering Complexity With ACE and Patterns*. Addison-Wesley, Boston, 2002.

[24] D. C. Schmidt, D. L. Levine, and S. Mungee. The Design and Performance of Real-Time Object Request Brokers. *Computer Communications*, 21(4):294–324, Apr. 1998.

[25] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2*. Wiley & Sons, New York, 2000.

[26] Srinivasan Ramani and Balabrishnan Dasarathy and Kishor S. Trivedi. Reliable Messaging Using the CORBA Notification Service. In *Proceedings of the $3^{rd}$ International Symposium on Distributed Objects and Applications (DOA 2001)*, Sep. 2001.

[27] Sun Microsystems. *Open Network Computing: Transport Independent RPC*, June 1995.

[28] A. Wollrath, R. Riggs, and J. Waldo. A Distributed Object Model for the Java System. *USENIX Computing Systems*, 9(4), November/December 1996.