

stated contract is satisfied.

- Facet references must behave properly with respect to component identity and navigation, as defined in sections Section 3.5.4 on page 39.

It is the intent of the specification that facet references are consistent throughout the life cycle of the component, although a crisp notion of consistency is almost impossible to define. In simple cases a component is realized as a set of servants explicitly bound together in a particular server. Each facet reference has a single servant at any point in time. All of the references returned by a particular `provide__name()` operation on a component instance should resolve to the “same” servant, i.e., a servant located by the same oid on a POA of the same name. In more complex implementations that may involve replication or dynamic load balancing, the logical identity of the component and its interfaces may not map directly onto distinct physical programming objects.

3.5.4 Navigation

Navigation among a component’s facets may be accomplished in the following ways:

- A client may navigate from any facet reference to the component that provides the reference via **CORBA::Object::get_component**.
- A client may navigate from the component interface to any facet using the generated **provide_name** operations on the component interface.
- A client may navigate from the component interface to any facet using the generic **provide_facet** operation on the **Navigation** interface (inherited by all component interfaces through **Components::ComponentBase**). Other operations on the **Navigation** interface (i.e., **provide_all_facets** and **provide_named_facets**) return multiple references, and can also be used for navigation. When using generic navigation operations on **Navigation**, facets are identified by string values that contain their declared names.
- A client may navigate from a facet interface that derives from the **Navigation** interface directly to any other facet on the same component, using **provide_facet**, **provide_all_facets**, and **provide_named_facets**.

The detailed descriptions of these mechanisms follow.

3.5.4.1 `get_component()`

The CORBA component specification extends the **CORBA::Object** pseudo interface with a single operation:

```

module CORBA {
    interface Object { // PIDL
        ...
        Object get_component ( );
    };
};

```

If the target object reference is itself a component reference (i.e., it denotes the component itself), the **get_component** operation returns the same reference (or another equivalent reference). If the target object reference is a facet reference the **get_component** operation returns an object reference for the component. If the target reference is neither a component reference nor a provided reference, **get_component** returns a nil reference.

Implementation of get_component

As with other operations on **CORBA::Object**, **get_component** is implemented as a request to the target object. Following the pattern of other **CORBA::Object** operations (i.e., **_interface**, **_is_a**, and **_non_existent**; see section 15.4.1.2 of the CORBA 2.3 specification), the operation name in GIOP request corresponding to **get_component** shall be “**_component**”.

Programming skeletons generated by the Component Implementation Framework for components and facets shall provide standard implementations for **get_component** (i.e., the **_component** request).

3.5.4.2 Component-specific provide operations

The **provide_name** operation implicitly defined by a **provides** declaration can be invoked to obtain a reference to the facet.

3.5.4.3 Navigation interface on the component

As described in Section 3.4 on page 35 all component interfaces implicitly inherit directly or indirectly from **ComponentBase**, which inherits from **Components::Navigation**. The definition of the **Components::Navigation** interface is as follows:

```

module Components {
    interface Navigation {
        valuetype FacetDescription {
            public RepositoryID InterfaceID;
            public FeatureName Name;
        };

        valuetype Facet : FacetDescription {
            public Object Ref;
        };

        typedef sequence<Facet> Facets;

        typedef sequence<FacetDescription>
            FacetDescriptions;

        Object provide_facet( in FeatureName name )
            raises (InvalidName);

        FacetDescriptions describe_facets();

        Facets provide_all_facets();

        Facets provide_named_facets(in NameList names)
            raises (InvalidName);

        boolean same_component( in Object ref );
    };
};

```

This interface provides generic navigation capabilities. It is inherited by all component interfaces, and may be optionally inherited by any interface that is explicitly designed to be a facet interface for a component. The descriptions of **Navigation** operations follow.

provide_facet

The **provide_facet** operation returns a reference to the facet denoted by the name parameter. The value of the name parameter must be identical to the name specified in the provides declaration. The valid names are defined by inherited closure of the actual type of the component, i.e., the names of facets of the component type and all of its inherited component types. If the value of the name parameter does not correspond to one of the component's facets, the **InvalidName** exception is raised.

describe_facets

The **describe_facets** operation returns a sequence containing descriptions of all of the facets provided by the target component. Each description is a **valuetype** containing the **RepositoryId** of the facet's interface and the name of the facet, expressed as an unscoped local name relative to the owning component's name scope. The order in which these descriptions occur in the sequence is not specified.

provide_all_facets

The **provide_all_facets** operation returns a sequence of value objects, each of which contains the **RepositoryId** of the facet interface and name of the facet, along with a reference to the facet. The sequence must contain descriptions and references for all of the facets in the component's inheritance hierarchy. The order in which these values occur in the sequence is not specified.

provide_named_facets

The **provide_named_facets** operation returns a sequence of described references (identical to the sequence returned by **provide_all_facets**), containing descriptions and references for the facets denoted by the **names** parameter. If any name in the **names** parameter is not a valid name for a provided interface on the component, the operation raises the **InvalidName** exception. The order of values in the returned sequence is not specified.

The **same_component** operation on **Navigation** is described in Section 3.5.5 on page 43.

3.5.4.4 Navigation interface on facet interfaces

Any interface that is designed to be used as a facet interface on a component may optionally inherit from the **Navigation** interface. When the navigation operations (i.e., **provide_facet**, **provide_all_facets**, **provide_named_facets**, and **describe_facets**) are invoked on the facet reference, the operations shall return the same results as if they had been invoked on the component interface that provided the target facet. The skeletons generated by the Component Implementation Framework will provide implementations of these operations that will delegate to the component interface.

This option allows navigation from one facet to another to be performed in a single request, rather than a pair of requests (to get the component reference and navigate from there to the desired facet). To illustrate, consider the following component definition:

```
module example {
  interface foo : Components::Navigation {... };
  interface bar { ... };
  component baz session {
    provides foo a;
    provides bar b;
  };
};
```

A client could navigate from a to b as follows:

```
foo myFoo;
// assume myFoo holds a reference to a foo provided by a baz
baz myBaz = bazHelper.narrow(myFoo.get_component());
bar myBar = myBaz.provide_b();
```

Or, it could navigate directly:

```
foo myFoo;
// assume myFoo holds a reference to a foo provided by a baz
bar myBar = barHelper.narrow(myFoo.provide_interface("b");
```

3.5.5 Provided References and Component Identity

The **same_component** operation on the **Navigation** interface allows clients to determine reliably whether two references belong to the same component instance, that is, whether the references are facets of or directly denote the same component instance. The component implementation is ultimately responsible for determining what the “same component instance” means. The skeletons generated by the Component Implementation Framework provide an implementation of **same_component**, where “same instance” is defined in terms of identity values supplied by the component implementation or the container in the container context. User-supplied implementations can provide different semantics.

If a facet interface inherits the **Navigation** interface, then the **same_component** operation on the provided interface should give the same results as the **same_component** operation on the component interface that owns the provided interface. The skeletons generated by the Component Implementation Framework provide an implementation of **same_component** for facets that inherit the **Navigation** interface.

3.5.6 Supported interfaces

A component definition may optionally support an interface, in a similar fashion to **valuetype**. When a component definition header includes a supports clause as follows:

```
component component_name supports interface_name { ... };
```

the component interface inherits both **ComponentBase** and the supported interface, as follows:

```
interface component_name
: Components::ComponentBase, interface_name { ... };
```

The component implementation must supply implementations of all of the operations defined on the supported interface. Clients must be able to narrow (or widen) a reference of the component interface type to the supported interface type. Clients must also be able to narrow a reference of type **ComponentBase** to the component’s supported interface type, if one exists.

For example, given the following IDL:

```

module M {
    interface I {
        void op();
    };
    component A supports I {
        provide I foo;
    };
    home AManager manages A {};
};

```

The AManager interface will be derived from HomeBase, supporting the create_component operation, which returns a reference of type ComponentBase. This reference must be able to be narrowed directly from ComponentBase to I:

```

// java
...
M.AManager aHome = ...; // get A's home
org.omg.Components.ComponentBase myComp =
aHome.create_component();
M.I myI = M.IHelper.narrow(myComp); // must succeed

```

A component may only support an interface directly if it does not inherit from another component type. The equivalent IDL inheritance for supported interfaces and derived component types is precise—interfaces for components that are derived directly or indirectly from a component definition that supports an interface will also inherit the supported interface.

For example, given the following IDL:

```

module M {
    interface I {
        void op();
    };
    component A supports I {
        provide I foo;
    };
    component B : A { ... };
    home BHome manages B {};
};

```

The equivalent IDL is:

```

module M {
    interface I {
        void op();
    };
    interface A :
    org.omg.Components::ComponentBase, I { ... };
    interface B : A { ... };
};

```

which allows the following usage:

```

M.BHome bHome = ... // get B's home
M.B myB = bHome.create();
myB.op(); // I's operations are supported directly on B's interface

```

The supports mechanism provides programming convenience for light-weight components that only need to implement a single operational interface. A client can invoke operations from the supported interface directly on the component reference, without narrowing or navigation:

```
M.A myA = aHome.create();
myA.op();
```

as opposed to

```
M.A myA = aHome.create();
M.I myI = myA.provide_foo();
myI.op();
```

or, assuming that the client has A's home, but doesn't statically know about A's interface or home interface:

```
org.omg.Components.HomeBase genericHome =
... // get A's home;
org.omg.Components.ComponentBase myComp =
genericHome.create_component();
```

```
M.I myI = M.IHelper.narrow(myComp);
myI.op();
```

as opposed to

```
org.omg.CORBA.Object obj = myComp.provide_interface("foo");
M.I myI = M.IHelper.narrow(obj);
myI.op();
```

This mechanism allows component-unaware clients to receive a reference to a component (passed as type CORBA::Object) and use the supported interface.

3.6 Receptacles

A component definition can describe the ability to accept object references upon which the component may invoke operations. When a component accepts an object reference in this manner, the relationship between the component and the referent object is called a *connection*; they are said to be *connected*. The conceptual point of connection is called a *receptacle*. A receptacle is an abstraction that is concretely manifested on a component as a set of operations for establishing and managing connections.

Receptacles are intended as a mechanical device for expressing a wide variety of relationships that may exist at higher levels of abstraction. As such, receptacles have no inherent higher-order semantics, such as implying ownership, or that certain operations will be transient across connections.

3.6.1 Syntax

The syntax for describing a receptacle is as follows:

<uses_dcl> ::= “uses” [“multiple”] <interface_type> <identifier>

A receptacle declaration comprises the following elements:

- The keyword **uses**.
- The optional keyword **multiple**. The presence of this keyword indicates that the receptacle may accept multiple object references simultaneously, and results in different operations on the component’s associated interface.
- An interface type, which must be either the keyword **Object** or a scoped name that denotes the interface type that the receptacle will accept. The scoped name must denote a previously-defined non-component interface type.
- An identifier that names the receptacle in the scope of the component.

3.6.2 Equivalent IDL

A **uses** declaration of the following form:

uses interface_type receptacle_name;

results in the following equivalent operations defined in the component interface:

```
void connect_receptacle_name ( in interface_type conxn )  
raises (  
    Components::AlreadyConnected,  
    Components::InvalidConnection  
);
```

```
interface_type disconnect_receptacle_name ( )  
raises ( Components::NoConnection );
```

```
interface_type get_connection_receptacle_name ( );
```

A **uses** declaration of the following form:

uses multiple interface_type receptacle_name;

results in the following equivalent operations defined in the component interface:


```

struct receptacle_nameConnection {
    interface_type objref;
    Components::Cookie ck;
};
sequence<receptacle_nameConnection> receptacle_nameConnections;

Components::Cookie
connect_receptacle_name ( in interface_type connection )
raises (
    Components::ExceededConnectionLimit,
    Components::InvalidConnection
);

interface_type disconnect_receptacle_name (
    in Components::Cookie ck
)
raises ( Components::InvalidConnection );

receptacle_nameConnections
get_connections_receptacle_name ( );

```

3.6.3 Behavior

3.6.3.1 Connect operations

Operations of the form **connect_receptacle_name** are implemented in part by the component implementor, and in part by generated code in the component servant framework. The responsibilities of the component implementation and servant framework for implementing connect operations are described in detail in Chapter 4. The receptacle holds a copy of the object reference passed as a parameter. The component may invoke operations on this reference according to its design. How and when the component invokes operations on the reference is entirely the prerogative of the component implementation. The receptacle will hold a copy of the reference until it is explicitly disconnected.

Simplex receptacles

If a receptacle's **uses** declaration does not include the optional **multiple** keyword, then only a single connection to the receptacle may exist at a given time. If a client invokes a connect operation when a connection already exists, the connection operation will raise the **AlreadyConnected** exception.

The component implementation may refuse to accept the connection for arbitrary reasons. If it does so, the connection operation will raise the **InvalidConnection** exception.

Multiplex receptacles

If a receptacle's **uses** declaration includes the optional **multiple** keyword, then multiple connections to the receptacle may exist simultaneously. The component implementation may choose to establish a limit on the number of simultaneous connections allowed. If an invocation of a connect operation attempts to exceed this limit, the operation will raise the **ExceededConnectionLimit** exception.

The component implementation may refuse to accept the connection for arbitrary reasons. If it does so, the connection operation will raise the **InvalidConnection** exception.

Connect operations for multiplex receptacles return values of type **Components::Cookie**. Cookie values are used to identify the connection for subsequent disconnect operations. Cookie values are generated by the receptacle implementation (the responsibility of the supplier of the component-enabled ORB, not the component implementor). Likewise, cookie equivalence is determined by the implementation of the receptacle implementation.

The client invoking connection operations is responsible for retaining cookie values and properly associating them with connected object references, if the client needs to subsequently disconnect specific references. Cookie values must be unique within the scope of the receptacle that created them. If a cookie value is passed to a disconnect operation on a different receptacle than that which created it, results are undefined.

Cookie values are described in detail in Section 3.6.3.4, "Cookie type".

Cookie values are required because object references cannot be reliably tested for equivalence.

3.6.3.2 Disconnect operations

Operations of the form **disconnect_receptacle_name** terminate the relationship between the component and the connected object reference.

Simplex receptacles

If a connection exists, the disconnect operation will return the connected object reference. If no connection exists, the operation will raise a **NoConnectionExists** exception.

Multiplex receptacles

The **disconnect_receptacle_name** operation of a multiplex receptacle takes a parameter of type **Components::Cookie**. The **ck** parameter must be a value previously returned by the **connect_receptacle_name** operation on the same receptacle. It is the responsibility of the client to associate cookies with object references they connect and disconnect. If the cookie value is not recognized by the receptacle implementation as being associated with an existing connection, the **disconnect_receptacle_name** operation raises an **InvalidConnection** exception.

3.6.3.3 *get_connection and get_connections operations*

Simplex receptacles

Simplex receptacles have operations named

get_connection_receptacle_name. If the receptacle is currently connected, this operation returns the connected object reference. If there is no current connection, the operation returns a nil object reference.

Multiplex receptacles

Multiplex receptacles have operations named

get_connections_receptacle_name. This operation returns a sequence of structures, where each structure contains a connected object reference and its associated cookie value. The sequence contains a description of all of the connections that exist at the time of the invocation. If there are no connections, the sequence length will be zero.

3.6.3.4 *Cookie type*

The **Cookie** valuetype is defined by the following IDL:

```
module Components {
    valuetype Cookie {
        private sequence<octet> cookieValue;
    };
};
```

Cookie values are created by multiplex receptacles, and are used to correlate a connect operation with a disconnect operation on multiplex receptacles.

Implementations of component-enabled ORBs may employ **valuetype** derived from **Cookie**, but any derived cookie types must be truncatable to **Cookie**, and the information preserved in the **cookieValue** octet sequence must be sufficient for the receptacle implementation to identify the cookie and its associated connected reference.

3.6.4 *Receptacles interface*

The **Receptacles** interface provides generic operations for connecting to a component's receptacles. The **ComponentBase** interface is derived from **Receptacles**. The **Receptacles** interfaces is defined by the following IDL:

```

module Components {

    valuetype ConnectionDescription {
        public Cookie ck;
        public Object objref;
    };

    typedef sequence<ConnectionDescription> ConnectedDescriptions;

    interface Receptacles {

        Cookie connect (
            in FeatureName name,
            in Object connection )
        raises (
            InvalidName,
            InvalidConnection,
            AlreadyConnected,
            ExceededConnectionLimit);

        void disconnect (
            in FeatureName name,
            in Cookie ck)
        raises (
            InvalidName,
            InvalidConnection,
            CookieRequired,
            NoConnection);

        ConnectionList get_connections (in FeatureName name)
        raises (InvalidName);

    };
};

```

connect

The **connect** operation connects the object reference specified by the **connection** parameter to the receptacle specified by the **name** parameter on the target component. If the specified receptacle is a multiplex receptacle, the operation returns a cookie value that can be used subsequently to disconnect the object reference. If the receptacle is a simplex receptacle, the return value is a null **valuetype**. The following exceptions may be raised:

- If the **name** parameter does not specify a valid receptacle name, then the **InvalidName** exception is raised.
- If the receptacle is a simplex receptacle and it is already connected, then the **AlreadyConnected** exception is raised.

- If the object reference in the **connection** parameter does not support the interface declared in the receptacle's **uses** statement, the **InvalidConnection** exception is raised.
- If the receptacle is a multiplex receptacle and the implementation-defined limit to the number of connections is exceeded, the **ExceededConnectionLimit** exception is raised.

disconnect

If the receptacle identified by the **name** parameter is a simplex receptacle, the operation will disassociate any object reference currently connected to the receptacle. The cookie value in the **ck** parameter is ignored. If the receptacle identified by the **name** parameter is a multiplex receptacle, the **disconnect** operation disassociates the object reference associated with the cookie value (i.e., the object reference that was connected by the operation that created the cookie value) from the receptacle. The following exceptions may be raised:

- If the **name** parameter does not specify a valid receptacle name, then the **InvalidName** exception is raised.
- If the receptacle is a simplex receptacle there is no current connection, then the **NoConnection** exception is raised.
- If the receptacle is a multiplex receptacle and the cookie value in the **ck** parameter does not denote an existing connection on the receptacle, the **InvalidConnection** exception is raised.
- If the receptacle is a multiplex receptacle and a null value is specified in the **ck** parameter, the **CookieRequired** exception is raised.

get_connections

The **get_connections** operation returns a sequence of **ConnectionDescription** structs. Each struct contains an object reference connected to the receptacle named in the **name** parameter, and a cookie value that denotes the connection. If the **name** parameter does not specify a valid receptacle name, then the **InvalidName** exception is raised.

3.7 Events

The CORBA component model supports a publish/subscribe event model. The event model for CORBA components is designed to be compatible with the OMG Notification Service, as defined in OMG document telecom/98-11-01, but it does not require that the Notification Service be used to implement the component event model. The data type of the events managed by the component event model is a compatible subset of **CosNotification::StructuredEvent** type. The interfaces exposed by the component event model provide a simplified API to a subset of Notification Service semantics.

3.7.1 Event service provided by container

Container implementations provide event services to components and their clients. Component implementations obtain event services from the container during initialization, and mediate client access to those event services. The container implementation is free to provide any mechanism that supports the required semantics. The container is responsible for configuring the mechanism and determining the specific quality of service and routing policies to be employed when delivering events. More detail is defined in Chapter 5, specifically section 5.3.8 and section 5.4.1.9.

3.7.2 Event Descriptions

An **emits** or **consumes** declaration in a component definition includes two string values, *event domain* and *event type*. These values correspond to the **domain_name** and **type_name** members of the **CosNotification::EventType** structure, and are used by the mechanism supplying event services to route events to interested consumers. Together, these values are called an *event description* for the purposes of the component event model.

When subscriber subscribes to an event source on a component, it will receive events whose descriptions match the description declared in the source's **emit** statement. From the subscriber's perspective, the event description of the source to which it subscribes is called its *subscription*.

When a publisher pushes events to an event sink on a component, the event description declared in the **consumes** statement for the sink is associated with the event. The event will be delivered only to subscribers with matching subscriptions.

3.7.3 Connectivity

The component event model provides general connectivity within a well-defined scope, such that a published event will be pushed to all subscribers in the scope whose subscriptions match the description contained in the event itself, subject to possible additional filtering imposed and managed by the container. This scope corresponds to a *channel* in the terminology of the Event and Notification services.

A container determines the scope of connectivity of event services provided to components that it manages.

3.7.4 Required Semantics

The subset of Notification semantics required is as follows:

- The ability to propagate event data in the form of sequences of name/value pairs that are structurally identical to the **CosNotification::FilterableEventBody** type. The component event model defines an equivalent type, **Components::EventData**.

- General connectivity within a well-defined scope, such that a published event will be pushed to all subscribers in the scope whose subscriptions match the description contained in the event description.
- The ability to support push semantics.
- Publish/subscribe semantics within a well-defined scope, equivalent to a notification channel.
- The ability to identify and route events based on a pair of string values, which are the equivalent of **domain_name** and **type_name** members of the **CosNotification::EventType** structure in the fixed header portion of the structured event, and to deliver to a subscriber only events whose values match the specification of the event source to which it subscribes. Implementations of the component event model may provide more sophisticated mechanisms for conditional event delivery, such as a full implementation of notification filtering. It is the responsibility of the container implementation to configure and manage these mechanisms.

3.7.5 Event Source

An event source on a component is the functional equivalent of a **CosNotification::StructuredPushSupplier** in Notification Service terminology, though it does not take that precise form. The syntax for describing an event source is as follows:

<emits_decl> ::= “emits” <event_descr> <identifier>

<event_descr> ::= <event_domain_spec> <event_type_spec>

<event_domain_spec> ::= “eventDomain” (<string_literal> | <scoped_name>)

<event_type_spec> ::= “eventType” (<string_literal> | <scoped_name>)

An event source declaration has the following elements:

- The keyword **emits**.
- An event domain specification, consisting of the keyword **eventDomain**, followed by a string value, which is either a string literal or a scoped name that denotes a previously defined string constant. The event domain corresponds to the **domain_name** member of the **CosNotification::EventType** structure in the fixed header portion of the structured event.
- An event type specification, consisting of the keyword **eventType**, followed by a string value, which is either a string literal or a scoped name that denotes a previously defined string constant. The event type does not denote an IDL type. It corresponds to the **type_name** member of the **CosNotification::EventType** structure in the fixed header portion of the structured event.
- An identifier that names the event source in the scope of the component.

For **emit** declarations of the form:

emits event_description source_name;

The component interface shall contain operations of the following form:

```
Components::Cookie
subscribe_source_name (
    in Components::Subscriber sub);
};

void unsubscribe_source_name (
    in Components::Cookie ck)
    raises ( Components::InvalidConnection );
};
```

A subscriber invokes **subscribe_source_name**, passing a reference to an object supporting the **Subscriber** interface. The operation returns a cookie that can be used subsequently to unsubscribe from the source, by invoking **unsubscribe_source_name**. If the cookie passed to the unsubscribe operation is not associated with an existing subscriber to the source, the operation raises an **InvalidConnection** exception.

3.7.6 Event Sinks

The syntax for describing an event sink is as follows:

```
<emits_decl> ::= "consumes" <event_descr> <identifier>

<event_descr> ::= <event_domain_spec> <event_type_spec>

<event_domain_spec> ::= "eventDomain" ( <string_literal> |
    <scoped_name> )

<event_type_spec> ::= "eventType" ( <string_literal> | <scoped_name> )
```

An event source declaration has the following elements:

- The keyword **consumes**.
- An event domain specification, consisting of the keyword **eventDomain**, followed by a string value, which is either a string literal or a scoped name that denotes a previously defined string constant. The event domain corresponds to the **domain_name** member of the **CosNotification::EventType** structure in the fixed header portion of the structured event.
- An event type specification, consisting of the keyword **eventType**, followed by a string value, which is either a string literal or a scoped name that denotes a previously defined string constant. The event type does not denote an IDL type. It corresponds to the **type_name** member of the **CosNotification::EventType** structure in the fixed header portion of the structured event.
- An identifier that names the event sink in the scope of the component.

For a **consumes** definition of the form:

consumes event_description sink_name;

The component interface shall contain operations of the following form:

void push_sink_name(in EventData evt);

Components::Subscriber get_sink_sink_name();

A publisher can push an event directly to the sink by invoking the **push_sink_name** operation. The **EventData** value will be associated with the event description declared on the sink's **consumes** statement, and delivered to subscribers with matching subscriptions in the same scope, subject to possible additional filtering imposed and managed by the container.

Alternatively, a publisher can obtain a reference from the component that supports the **Components::Subscriber** interface. The publisher can push events to the sink through this interface by invoking the **push** method on the reference. Invoking the push method on this reference is equivalent to invoking the **push_sink_name** operation.

3.7.7 Subscriber interface

The component event model defines the **Subscriber** interface as follows:

```
module Components {

    struct Property {
        PropertyName name;
        PropertyValue value;
    };

    typedef sequence<Property> EventData;

    interface Subscriber {
        void push(in EventData evt);
    };

};
```

The **Subscriber** interface is implemented by clients in order to receive events from component event sources. The component event source will invoke **push** to propagate events to the client. The source will only propagate events matching the event description on the source's **emits** declaration.

The **Subscriber** interface is also supported by component event sinks. When a client invokes **push** on a sink's **Subscriber** interface, the event description on the sink's **consumes** declaration will be associated with the event, and it will be delivered to matching subscribers in the same scope.