An Overview of OMG CORBA Event Services

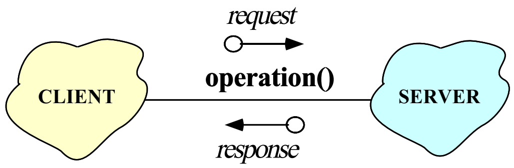
Douglas C. Schmidt

Washington University, St. Louis

http://www.cs.wustl.edu/∼schmidt/

schmidt@cs.wustl.edu

Event Services

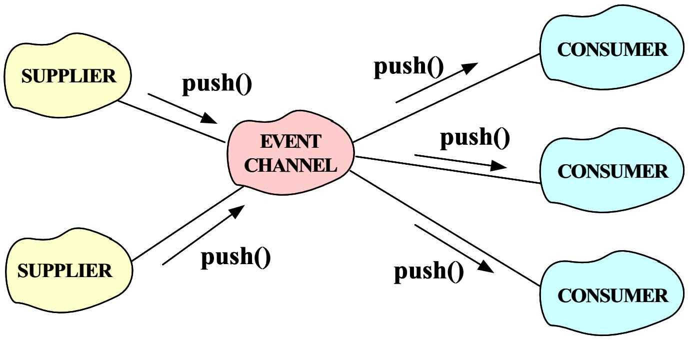


- Standard CORBA method invocations result in synchronous execution of an operation provided by an object
  - Both requestor (client) and provider (server) must be present
  - Client blocks until operation returns
  - Only supports uni-cast communication

OMG Event Services

- For many applications, a more decoupled communication model between objects is required
  - *i.e.*, asynchronous communication with multiple *suppliers* and *consumers*
- OMG defines a set of event service interfaces that enable decoupled, asynchronous communication between objects
- The OMG model is based on the "publish/subscribe" paradigm
  - The basic model is also useful for more sophisticated types of event services
    - *e.g.*, filtering and event correlation

Common Event Service Collaborations



- Note: no (implicit) responses

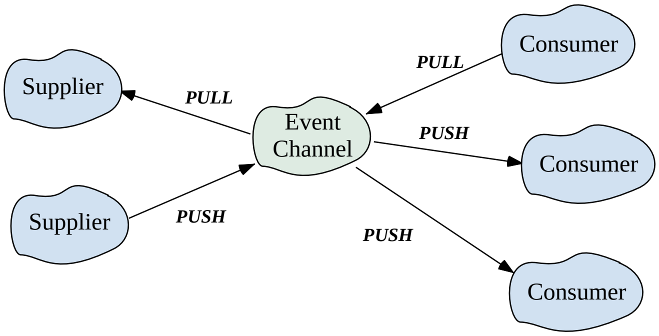
Benefits of the OMG Event Service

- *Anonymous consumers/suppliers*
  - Publish and subscribe model
- *Group communication*
  - Supplier(s) to consumer(s)
- *Decoupled communication*
  - Asynchronous delivery
- *Abstraction for distribution*
  - Can help draw the lines of distribution in the system
- *Abstraction for concurrency*
  - Can facilitate concurrent event handling

Event Service Participants

- The OMG event service defines three roles
  1. *The Supplier role*
     - Suppliers generate event data
  2. *The Consumer role*
     - Consumers process event data
  3. *Event Channel*
     - A "mediator" that encapsulates the queueing and propagation semantics
- Event data are communicated between suppliers and consumers by issuing standard CORBA (twoway) requests
  - Standard CORBA naming and object activation mechanisms can also be used

Structure and Interaction Among Participants

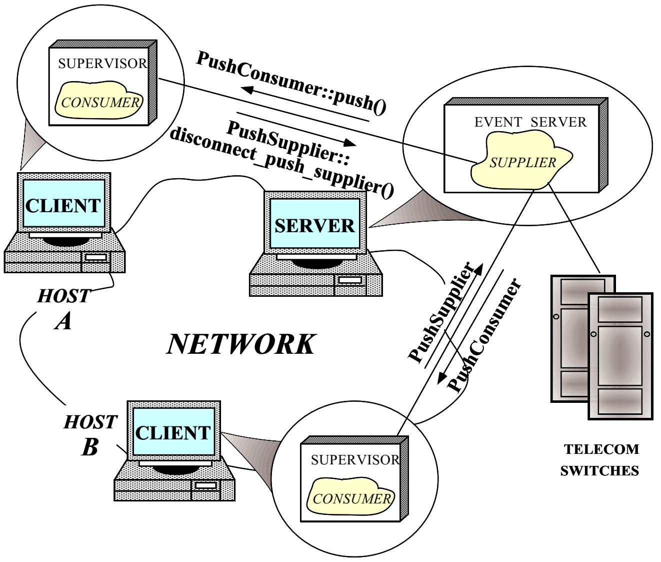


- Note both *Push* and *Pull* models supported

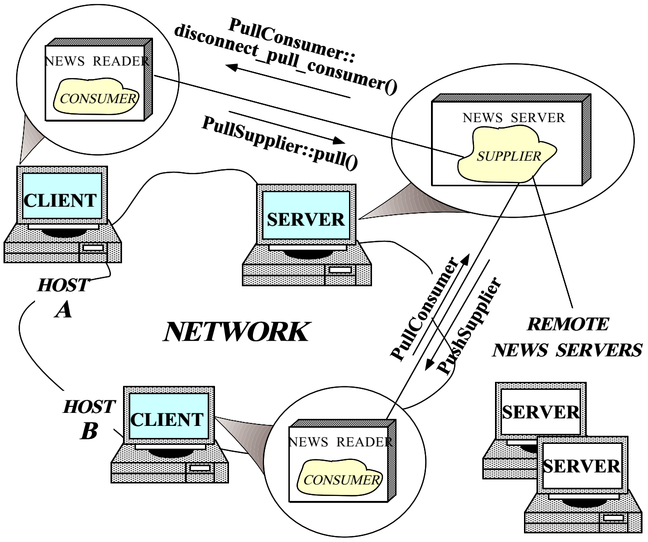
The Push and Pull Models

- There are two general approaches for initiating event communication between suppliers and consumers
  1. *The push model*
     - The push model allows a supplier of events to initiate the transfer of the event data to consumers
     - Note the *supplier* takes the initiative in the push model
  2. *The pull model*
     - The pull model allows a consumer of events to request event data from a supplier
     - Note the *consumer* takes the initiative in the pull model

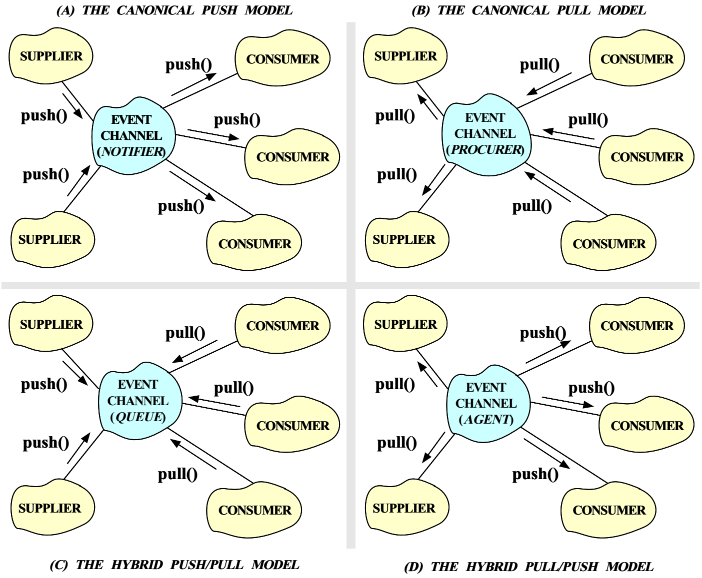
The Push Model



The Pull Model



Communication Models for Event Channels



Generic and Typed Event Communication

- There are two orthogonal approaches that OMG event-based communication may take:

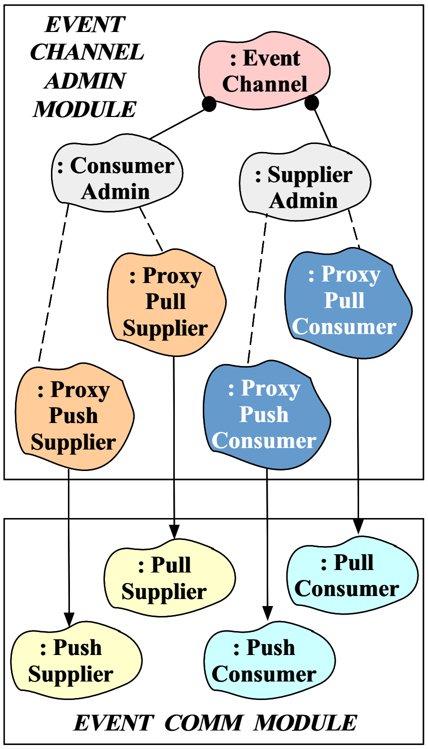
  1. *Generic*

     - All communication is by means of generic `push` or `pull` operations
     - These operations involve single parameters or return values that package all the events into a generic CORBA `any` data structure

  2. *Typed*

     - In the typed case, communication is via operations defined in OMG IDL
     - Event data is passed by means of typed parameters, which can be defined in any desired manner

Event Service Class Structure



The EventComm Module

- The *event communication module* `EventComm` illustrated below defines a set of CORBA interfaces for event-style communication

```
module CosEventComm {
  exception Disconnected {};

  interface PushConsumer {
    void push (in any data) raises (Disconnected);
    void disconnect_push_consumer ();
  };

  interface PushSupplier {
    void disconnect_push_supplier ();
  };

  interface PullSupplier {
    any pull() raises (Disconnected);
    any try_pull() (out boolean has_event)
      raises (Disconnected);
    void disconnect_pull_supplier ();
  };

  interface PullConsumer {
    void disconnect_pull_consumer ();
  };
```

The PushConsumer Interface

- A push consumer implements the `PushConsumer` interface to receive event data from a supplier

```
interface PushConsumer
{
  void push (in any data) raises (Disconnected);
  void disconnect_push_consumer ();
};
```

- A supplier communicates event data to the consumer by invoking the `push` operation on an object reference and passing the event data as a parameter

- The `disconnect_push_consumer` operation terminates the event communication and releases resources

The PushSupplier Interface

- A push supplier implements the `PushSupplier` interface to disconnect from a supplier

```
interface PushSupplier
{
  void disconnect_push_supplier ();
};
```

- The `disconnect_push_supplier` operation terminates the event communication and releases resources

The PullSupplier Interface

- A pull supplier implements the `PullSupplier` interface to transmit event data to a consumer

```
interface PullSupplier {
  any pull() raises (Disconnected);
  any try_pull() (out boolean has_event)
    raises (Disconnected);
  void disconnect_pull_supplier ();
};
```

- A consumer requests event data from the supplier by invoking either the `pull` operation (blocking) or the `try_pull` operation (non-blocking) on the supplier
- The `disconnect_pull_supplier` operation terminates event communication and releases resources

The PullConsumer Interface

- A pull consumer implements the `PullConsumer` interface to disconnect from a consumer

```
interface PullConsumer
{
  void disconnect_pull_consumer ();
};
```

- The `disconnect_pull_consumer` operation terminates the event communication and releases resources

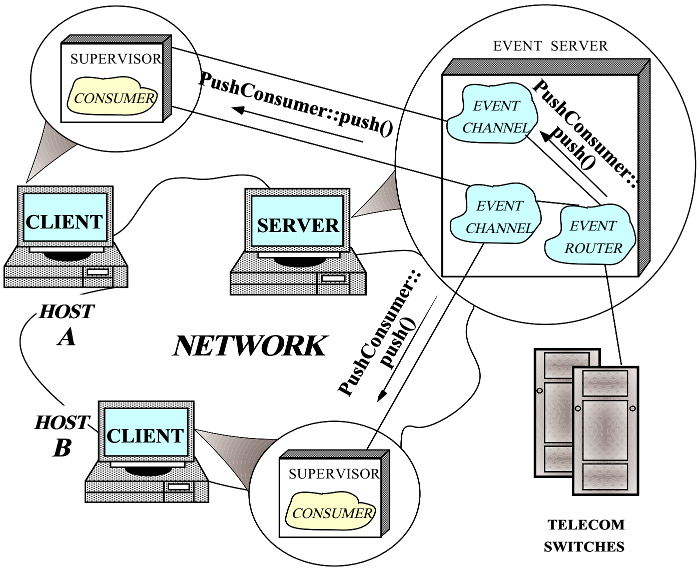
Event Channel Overview

- In addition to consumers and suppliers, OMG event services also have the notion of an *event channel*
  - An event channel is an object that allows multiple suppliers to communicate with multiple consumers in a highly decoupled, asynchronous manner
- An event channel is both a consumer and supplier of event data that it receives
  - In its simplest form, an event channel acts as “broadcast repeater”

Event Channel Overview (cont’d)

- Event channels are standard CORBA objects, and communication with an event channel is accomplished using standard CORBA requests
- However, an event channel need not supply the incoming event data to its consumer(s) at the same time it consumes data from its supplier(s)
  - *i.e.*, it may buffer data

Event Channel Use-case



Push-Style Communication with an Event Channel

- The supplier pushes event data to an event channel
- The event channel, in turn, pushes event data to all consumers
  - Note that an event channel need not make any complex routing decision, *e.g.*, it can simply deliver the data to all consumers
  - More complex semantics are also possible, of course

Pull-Style Communication with an Event Channel

- The consumer pulls event data from the event channel
- The event channel, in turn, pulls event data from the suppliers
  - This can be optimized by adding a queueing mechanism in the Event Channel

Multiple Consumers and Multiple Suppliers

- An event channel may provide many-to-many communication
- The channel consumes events from one or more suppliers, and supplies events to one or more consumers
- Subject to the quality of service of a particular implementation, an event channel provides an event to all consumers
- An event channel can support consumers and suppliers that use different communication models

Mixed-style Communication with an Event Channel

- An event channel can communicate with a supplier using one style of communication, and communicate with a consumer using a different style of communication
- Note that how long an event channel must buffer events is defined as a "quality of implementation" issue

---

Event Channel Administration

- An event channel is built up incrementally
  - *i.e.*, when a channel is created no suppliers or consumers are connected
- An `EventChannelFactory` object is used to return an object reference that supports the `EventChannel` interface
- The `EventChannel` interface defines three administrative operations:
  1. **`ConsumerAdmin`** → a factory for adding consumers
  2. **`SupplierAdmin`** → a factory for adding suppliers
  3. An operation for destroying the channel

Event Channel Administration (cont'd)

- The `ConsumerAdmin` factory operation returns a *proxy supplier*
  - A proxy supplier is similar to a normal supplier (in fact, it inherits the supplier interface)
  - However, it includes a method for connecting a consumer to the proxy supplier
- The `SupplierAdmin` factory operation returns a *proxy consumer*
  - A proxy consumer is similar to a normal consumer (in fact it inherits the interface of a consumer)
  - However, it includes an additional method for connecting a supplier to the proxy consumer

---

Event Channel Administration (cont'd)

- Registering a supplier with an event channel is a two-step process
  1. An event-generating application first obtains a proxy consumer from a channel
  2. It then "connects" to the proxy consumer by providing it with a supplier object reference
- Likewise, registering a consumer with an event channel is also a two-step process
  1. An event-receiving application first obtains a proxy supplier from a channel
  2. It then "connects" to the proxy supplier by providing it with a consumer object reference

Event Channel Administration (cont'd)

- The reason for the two-step registration process is to support composing event channels created by an *external agent*

- Such an agent would compose two channels by obtaining a proxy supplier from one (via the channel's `SupplierAdmin` factory)

- It would then obtain a proxy consumer from the other channel (via the channel's `ConsumerAdmin` factory)

- Finally, it would pass each of the proxy object references to the other channel as part of their connection procedure

---

The EventChannelAdmin Module

- The `EventChannelAdmin` module defines the interfaces for making connections between suppliers and consumers

```
#include "EventComm.idl"
module CosEventChannelAdmin {

  exception AlreadyConnected {};
  exception TypeError {};

  interface ProxyPushConsumer
    : CosEventComm::PushConsumer
  {
    void connect_push_supplier
      (in CosEventComm::PushSupplier push_supplier)
      raises (AlreadyConnected);
  };

  interface ProxyPullSupplier
    : CosEventComm::PullSupplier
  {
    void connect_pull_consumer
      (in CosEventComm::PullConsumer pull_consumer)
      raises (AlreadyConnected);
  };
```

The EventChannelAdmin Module (cont'd)

- `interface EventChannelAdmin` (cont'd)

```
interface ProxyPullConsumer
  : CosEventComm::PullConsumer
{
  void connect_pull_consumer
    (in CosEventComm::PullSupplier pull_supplier)
    raises (AlreadyConnected, TypeError);
};

interface ProxyPushSupplier
  : CosEventComm::PushSupplier
{
  void connect_push_consumer
    (in CosEventComm::PushConsumer push_consumer)
    raises (AlreadyConnected, TypeError);
};
```

The EventChannelAdmin Module (cont'd)

- `interface EventChannelAdmin` (cont'd)

```
  interface ConsumerAdmin {
    ProxyPushSupplier obtain_push_supplier ();
    ProxyPullSupplier obtain_pull_supplier ();
  };

  interface SupplierAdmin {
    ProxyPushConsumer obtain_push_consumer ();
    ProxyPullConsumer obtain_pull_consumer ();
  };

  interface EventChannel {
    ConsumerAdmin for_consumers ();
    SupplierAdmin for_suppliers ();
    void destroy ();
  };
};
```

The EventChannel Interface

- The `EventChannel` interface defines three administrative operations
  1. Adding consumers
  2. Adding suppliers
  3. Destroying the channel
- *e.g.*,

```
interface EventChannel {
  ConsumerAdmin for_consumers ();
  SupplierAdmin for_suppliers ();
  void destroy ();
};
```

The EventChannel Interface (cont'd)

- Consumer administration and supplier administration are defined as separate objects so that the creator of the channel can control the addition of suppliers and consumers, *e.g.*,
  - An event channel creator might wish to be the sole supplier of event data, but might allow many consumers to be connected to the channel
  - In this case, the creater would simply export the `ConsumerAdmin` object

```
interface Document
{
  ConsumerAdmin title_changed ();
};
```

The EventChannel Interface (cont'd)

- Any object that possesses an object reference that supports the `EventChannel` interface can perform the following operations
  - The `ConsumerAdmin` interface allows consumers to be connected to an event channel
    - The `for_consumers` operation returns an object reference that supports the `ConsumerAdmin` interface
  - The `SupplierAdmin` interface allows suppliers to be connected to an event channel
    - The `for_suppliers` operation returns an object reference that supports the `SupplierAdmin` interface
  - The `destroy` operation destroys the event channel

The ConsumerAdmin Interface

- The `ConsumerAdmin` interface defines the first step for connecting consumers to an event channel
  - Clients use this interface to obtain proxy suppliers

```
interface ConsumerAdmin {
  ProxyPushSupplier obtain_push_supplier ();
  ProxyPullSupplier obtain_pull_supplier ();
};
```

- The `obtain_push_supplier` operation returns a `ProxyPushSupplier` object that may be used to connect a push-style consumer
- The `obtain_pull_supplier` operation returns a `ProxyPullSupplier` object that may be used to connect a pull-style consumer

The SupplierAdmin Interface

- The `SupplierAdmin` interface defines the first step for connecting suppliers to an event channel
  - Servers use it to obtain proxy consumers

```
interface SupplierAdmin {
  ProxyPushConsumer obtain_push_consumer ();
  ProxyPullConsumer obtain_pull_consumer ();
};
```

- The `obtain_push_consumer` operation returns a `ProxyPushConsumer` object that may be used to connect a push-style supplier

- The `obtain_pull_consumer` operation returns a `ProxyPullConsumer` object that may be used to connect a pull-style supplier

The ProxyPushConsumer Interface

- The `ProxyPushConsumer` interface defines the second step for connecting push suppliers to an event channel

```
interface ProxyPushConsumer
  : CosEventComm::PushConsumer
{
  void connect_push_supplier
    (in CosEventComm::PushSupplier push_supplier)
    raises (AlreadyConnected);
};
```

The ProxyPushConsumer Interface (cont'd)

- A nil object reference may be passed to the `connect_push_supplier` operation
  - If so, a channel can't call `disconnect_push_supplier` on the supplier
  - Therefore, the supplier may be disconnected from the channel without being informed

- If the `ProxyPushConsumer` is already connected to a `PushSupplier`, then the exception `AlreadyConnected` is raised

The ProxyPullSupplier Interface

- The `ProxyPullSupplier` interface defines the second step for connecting pull consumers to an event channel

```
interface ProxyPullSupplier
  : CosEventComm::PullSupplier
{
  void connect_pull_consumer
    (in CosEventComm::PullConsumer pull_consumer)
    raises (AlreadyConnected);
};
```

The ProxyPullSupplier Interface (cont'd)

- A nil object reference may be passed to the `connect_pull_consumer` operation; if so a channel can't call `disconnect_pull_consumer` on the consumer
  - Therefore, the consumer may be disconnected from the channel without being informed
- If the `ProxyPullSupplier` is already connected to a `PullConsumer`, then the exception `AlreadyConnected` is raised

---

The ProxyPullConsumer Interface

- The `ProxyPullConsumer` interface defines the second step for connecting pull suppliers to an event channel

```
interface ProxyPullConsumer
  : CosEventComm::PullConsumer
{
  void connect_pull_consumer
    (in CosEventComm::PullSupplier pull_supplier)
    raises (AlreadyConnected, TypeError);
};
```

The ProxyPullConsumer Interface (cont'd)

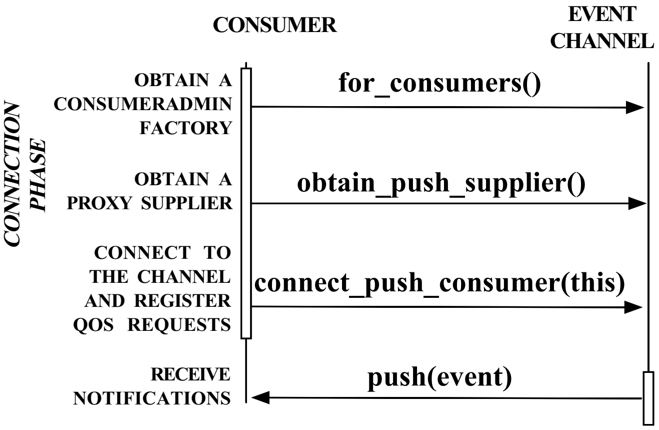
- Implementations should raise the standard BAD_PARAM exception if a nil object reference is passed to `connect_pull_supplier`
- If the `ProxyPullConsumer` is already connected to a `PullSupplier`, then the exception `AlreadyConnected` is raised
- An implementation of a `ProxyPullConsumer` may put additional requirements on the interface supported by the pull supplier
  - If the pull supplier does not meet those requirements, the `ProxyPullConsumer` raises the exception `TypeError`

The ProxyPushSupplier Interface

- The `ProxyPushSupplier` interface defines the second step for connecting push consumers to the event channel

```
interface ProxyPushSupplier
  : CosEventComm::PushSupplier
{
  void connect_push_consumer
    (in CosEventComm::PushConsumer push_consumer)
    raises (AlreadyConnected, TypeError);
};
```

Connecting a Consumer to an Event Channel


CONSUMER
EVENT CHANNEL
CONNECTION PHASE
OBTAIN A CONSUMERADMIN FACTORY
for_consumers()
OBTAIN A PROXY SUPPLIER
obtain_push_supplier()
CONNECT TO THE CHANNEL AND REGISTER QOS REQUESTS
connect_push_consumer(this)
RECEIVE NOTIFICATIONS
push(event)

The ProxyPushSupplier Interface (cont'd)

- Implementations should raise the standard BAD_PARAM exception if a nil object reference is passed to `connect_push_supplier`
- If the `ProxyPushSupplier` is already connected to a `PullConsumer`, then the exception `AlreadyConnected` is raised
- An implementation of a `ProxyPushSupplier` may put additional requirements on the interface supported by the push consumer
  - If the push consumer does not meet those requirements, the `ProxyPushSupplier` raises the `TypeError` exception

Typed Event Communication

- The preceeding discussion of OMG event services utilizes the properties of the CORBA **any** type to enable *generic* communication of event data
  - The **any** type supports extremely flexible models of interworking

```
struct any {
  typeCode *_type;
  void *_value;
  // ...
};
```

- However, it may be inconvenient or inefficient for use **any** in certain types of applications
  - In many applications, it is more appropriate to use *typed* communication between suppliers and consumers
  - Therefore, OMG also provides a parallel set of `TypedEventComm` and `TypedEventChannelAdmin` interfaces

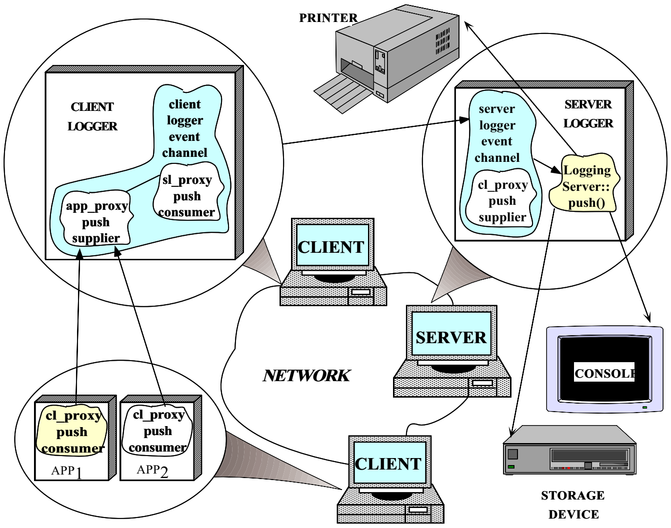
Composing Event Channels and Filtering

- The event channel administration operations defined in the `EventChannelAdmin` interface support the composition of event channels
  - *i.e.*, one event channel can consume events supplied by another
- This architecture allows the implementation of an event channel that filters the events supplied by another
  - *e.g.*, filtering based on event type

Policies for Finding Event Channels

- The OMG event service does *not* establish policies for locating event channels
  - Finding a service is orthogonal to using the service
- Higher levels of software may define policies for locating and using event channels
  - *i.e.*, higher layers will dictate when an event channel is created and how references to the event channel are obtained
- By representing the event channel as a CORBA object, it has all of the properties that apply to objects
  - *i.e.*, name servers, object locator mechanisms, marshalling, etc.

Example



- Distributed logging facility

Application Logger Interface

- Module specifying interface for client application logging

```
module Logger {

  enum Log_Priority {
      LOG_DEBUG,   // Debugging messages
      LOG_WARNING, // Warning messages
      LOG_ERROR,   // Errors
      LOG_EMERG,   // A panic condition
  };

  struct Log_Record {
    Log_Priority type;// Type of logging record
    long time_stamp;  // Time logging record generated
    long pid;       // Application process id
    string msg_data;  // Log record data
  };
```

Application Logger Interface (cont'd)

- Logging interface (cont'd)

```
  exception Invalid_Record { };

  interface Log
  {
    // Main method for logging a Log_Record
    void log (in Log_Record log_rec)
      raises (Invalid_Record);
  };
};
```

Client Application Logging

- Client application obtains object reference to Logger object and performs logging calls

```
using namespace Logger;

// Find any Logger implementation.
Log_var logger =
  bind_service<Log> ("Logger");

Log_Record log_rec;

// Initialize the log_record
log_rec.type = Logger::LOG_DEBUG;
log_rec.time_stamp = ::time (0);
// ...

try {
  logger->log (log_rec);
}
catch (Logger::Invalid_Record &) {
  // ...
}
```

Client Logger Interface

- Interface for the Client Logger

```
interface Client_Logger {
  SupplierAdmin for_suppliers ();
};
```

- The Client logger is typically located on the same host as the applications
  - It performs a "multiplexing service"
- However, it could also be located on another host within a network
- Regardless of location, the CORBA Name Service mechanism will find the appropriate object reference

---

Server Logger Interface

- Interface for the Server Logger

```
interface Server_Logger {
SupplierAdmin for_suppliers ();
};
```

- The Server Logger may be located anywhere in a network
  - Including *co-located* or *replicated*
- The CORBA locator mechanism is responsible for determining where a Server Logger resides

---

Application Logger Interface Implementation

- Implement client's logging interface

```
class My_Log : public virtual Logger::LogBOAImpl {
public:
  My_Log (void) {
    // Locate the Client Logger event channel.
    Client_Logger_var cl =
      bind_service<Client_Logger> ("Client_Logger");
    SupplierAdmin_var supplier_admin =
      cl->for_suppliers ();
    this->cl_proxy_push_consumer_ =
      supplier_admin->obtain_push_consumer ();
    // Don't allow two-way communication or disconnects.
    this->cl_proxy_push_consumer->
      connect_push_supplier (CORBA::nil ());
  }

  void log (const Logger::Log_Record &log_rec) {
    CORBA::any msg (TC_LOG_RECORD, &log_rec);
    // Push this to the Client Logger channel.
    this->cl_proxy_push_consumer_->push (msg);
  }
private:
  ProxyPushConsumer_var cl_proxy_push_consumer_;
};
```

Server Logger PushConsumer Implementation

- This is the final destination of an application's `log` operation

```
class My_Logging_Server
  : public virtual CosEventComm::PushConsumer {
public:
  My_Logging_Server (void):
    log_type_ (new CORBA::typeCode (TC_LOG_RECORD)) {}
  ~My_Logging_Server (void) { delete this->log_type_; }

  virtual void push (any *msg) {
    if (msg->_type->kind () == tk_struct) {
      any *struct_type = msg->_type.parameter (0);
      if (struct_type->_type->equal (this->log_type_)) {
        Logger::Log_Record *log_rec =
          static_cast <Logger::Log_Record *>
            (struct_type->_value);
        clog << log_rec.msg_data << ....;
        return;
      }
    } // otherwise there's an error...
  }
private:
  CORBA::typeCode *log_type_;
```

Client Logger Implementation

- Implementation of the `SupplierAdmin` factory

```
class My_Client_Logger
{
public:
  SupplierAdmin_ptr for_suppliers (void) {
    make_cl_channel ();
    return make_supplier_admin ();
  }

  void make_cl_channel (void);
  SupplierAdmin_ptr make_supplier_admin (void);

private:
  // Proxy to our EventChannel.
  EventChannel_ptr cl_channel_;

  // Proxy to the Server's Event Channel.
  Server_Logger_ptr sl_channel_proxy_;
}
```

Client Logger Implementation (cont'd)

- Create the Client Logger's Event Channel

```
void My_Client_Logger::make_cl_channel (void)
{
  // Magically create an EventChannelFactory and
  // create our Client_Logger EventChannel.
  EventChannelFactory_var factory = ...;
  cl_channel_ =
    factory->create_event_channel ();

  // Get a proxy to the Server Logger.
  sl_channel_proxy_ =
    bind_service<Server_Logger> ("Server_Logger");
}
```

- Note that we would probably use a "FactoryFinder" from the COSS Life Cycle specification to obtain our `EventChannelFactory`

Client Logger Implementation (cont'd)

- Return the `SupplierAdmin`

```
SupplierAdmin_ptr
My_Client_Logger::make_supplier_admin (void)
{
  // Obtain all the necessary proxies.
  ConsumerAdmin_var consumer_admin =
    cl_channel_->for_consumers ();
  ProxyPushSupplier_var app_proxy_push_supplier =
    consumer_admin->obtain_push_supplier ();
  SupplierAdmin_var supplier_admin =
    sl_channel_proxy_->for_suppliers ();
  ProxyPushConsumer_var sl_proxy_push_consumer =
    supplier_admin->obtain_push_consumer();

  // Use double-dispatch to connect everything together.
  sl_proxy_push_consumer->
    connect_push_supplier (app_proxy_push_supplier);
  app_proxy_push_supplier->
    connect_push_consumer (sl_proxy_push_consumer);

  // Return connected supplier admin.
  return cl_channel_->for_suppliers ();
}
```

Server Logger Implementation

- Implementation of Server Logger `SupplierAdmin` factory

```
class My_Server_Logger
{
public:
  SupplierAdmin_ptr for_suppliers (void) {
    make_sl_channel ();
    return make_supplier_admin ();
  }

  void make_sl_channel (void);
  SupplierAdmin_ptr make_supplier_admin (void);

private:
  // Proxy to our EventChannel.
  EventChannel_var sl_channel_;

  // Implementation of the actual PushConsumer.
  PushConsumer_var server_logger_;
};
```

Server Logger Implementation (cont'd)

- Create the Server Logger's Event Channel

```
void My_Server_Logger::make_sl_channel (void)
{
  // Magically create an EventChannelFactory and
  // create our Client_Logger EventChannel.
  EventChannelFactory_var factory = ...;

  sl_channel_ = factory->create_eventchannel ();
}
```

- Note that we would probably use a "FactoryFinder" from the COSS Life Cycle specification to obtain our `EventChannelFactory`

Server Logger Implementation (cont'd)

- Return the `SupplierAdmin`

```
SupplierAdmin_ptr
My_Server_Logger::make_supplier_admin (void) {
  // Obtain proxies to the Supplier/Consumer
  // factories and Proxies
  SupplierAdmin_var supplier_admin =
    sl_channel_->for_suppliers ();
  ConsumerAdmin_var consumer_admin =
    sl_channel_->for_consumers ();
  ProxyPushSupplier_var cl_proxy_push_supplier =
    consumer_admin->obtain_push_supplier ();

  // Initialize the PushConsumer implementation.
  server_logger_ = new My_Logging_Server;

  // Double-dispatch to connect everything together.
  cl_proxy_push_supplier->
    connect_push_supplier (server_logger);

  return supplier_admin;
}
```

Advanced Event Channel Services

- Note that a simple event channel implementation contains no real routing intelligence
  - *i.e.*, it simply forwards all events it receives from supplier to consumer (assuming the push model is used)
- A more sophisticated event channel implementation could provide a type of "event router"
  - This router would selectively decide which event channel(s) receive which events
- Even more sophisticated schemes could provide additional semantics
  - *e.g.*, filtering, correlation, persistence, fault tolerance, real-time scheduling, etc.
  - See www.cs.wustl.edu/∼schmidt/oopsla.ps.gz

Case Study: Real-time Event Channels

- Asynchronous messaging and group communication are important for real-time applications
  - *e.g.*, avionics mission control systems, telecom gateways, etc.
- The following example presents our OO architecture for CORBA *Real-time Event Channels*
- Focus is on *design patterns* and *reusable framework* components

Real-time Issues Not Addressed by COS Event Services

- *Deadlines*
  - Real-time tasks with data and event dependencies require predictable event notifications
    - *e.g.*, consumers must receive events in time to meet deadlines
- *Scheduling*
  - Real-time systems must guarantee that higher priority tasks are notified before lower priority tasks
    - *e.g.*, policies for event propagation
- *Periodic Tasks*
  - Periodic tasks must always run at certain intervals
    - *e.g.*, timers and rate groups

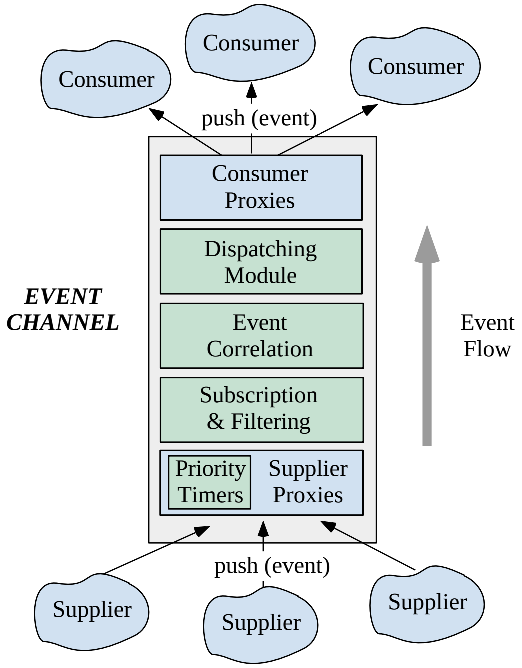
Open vs. Closed Systems

- *Definitions*
  - *Open systems* are systems designed to work correctly even when they have no idea of all other components in the system
    - *e.g.*, WWW browsers running Java Applets
  - *Closed systems* are ones that know how all the other components in the system behave
    - *e.g.*, existing RT avionics systems
- *Challenge*
  - Identify the structure and boundaries of the *open* and *closed* aspects for Real-time avionics system
  - Central issues are:
    - *Trust*
    - *Dependencies*
    - *Time to run*

Enhancing COS Event Services for Real-time Systems

- To enhance the COS Event Services for Real-time we've defined:
  1. *Real-time scheduling policies*
  2. *Real-time dispatching*
  3. *Quality of Service interfaces*
  4. *Flexible concurrency strategies*
  5. *Event filtering and correlation*
- Goal – "as close to the COS specification as possible, but no closer"

RT Event Service Architecture

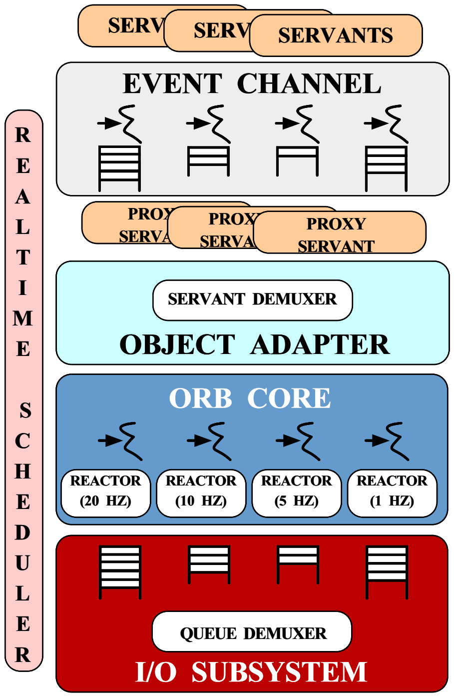


Consumer
Consumer
Consumer
push (event)
Consumer Proxies
Dispatching Module
Event Correlation
Subscription & Filtering
Priority Timers
Supplier Proxies
EVENT CHANNEL
Event Flow
push (event)
Supplier
Supplier
Supplier

Real-time Scheduling Policies

- *Problem*
  - Order in which events are forwarded by COS Event Channels is not defined by the specification
- *Solution*
  - An RT event channel must integrate with system-wide scheduling policies
    - *e.g.*, rate monotonic
  - Achieving this requires specific information from Suppliers and Consumers
    - *e.g.*, period, worst-case execution time, etc.

Real-time RTEC Scheduler

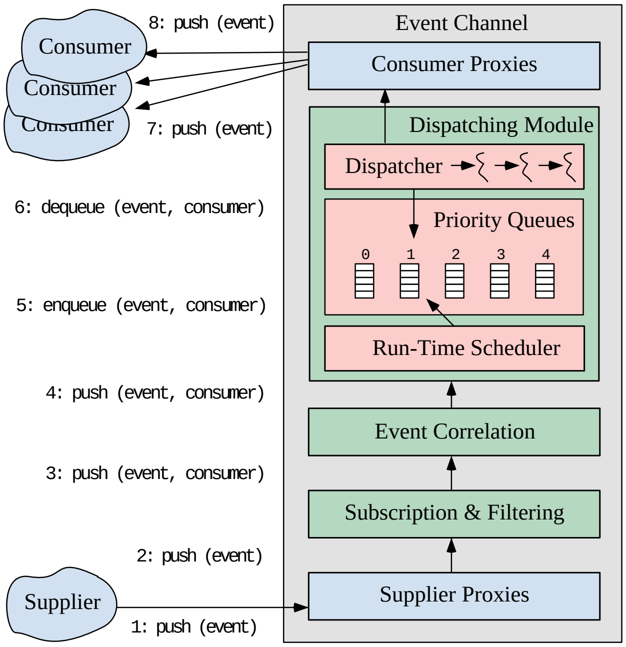


SERV
SERV
SERVANTS
EVENT CHANNEL
PROX SERVA
PROX SERVA
PROXY SERVANT
SERVANT DEMUXER
OBJECT ADAPTER
ORB CORE
REACTOR (20 HZ)
REACTOR (10 HZ)
REACTOR (5 HZ)
REACTOR (1 HZ)
QUEUE DEMUXER
I/O SUBSYSTEM
REAL TIME SCHEDULER

Real-time Dispatching Mechanisms

- *Problem*
  - To ensure deadlines are met, Event Channel must always dispatch highest priority event within a small, bounded amount of time
- *Solution*
  - Create a Dispatcher Module that maintains a queue for every Consumer priority level
  - The Dispatcher Module always dispatches events in higher priority queues before lower priority queues
  - Various types of *preemption* are supported

Real-time Dispatcher



Quality of Service Interfaces

- *Problem*
  - Suppliers and Consumers must relay their quality of service (QoS) requirements to the channel
  - Event Service mechanisms for coordinating scheduling data should integrate with global scheduling mechanism
- *Solution*
  - Define a system-wide Execution Model that provides abstractions for obtaining threads of control and publishing scheduling characteristics
  - All components in the system must either:
    - Use the Execution Model directly, or
    - Use Adapters to integrate 'off-the-shelf' toolkits into the Execution Model

---

Execution Model Definitions

- *Operation* → work that needs to be done in reponse to an event
  - *e.g.*, I/O, timer, method call
  - Typically encapsulated by an object
- *RT Operation* – work that needs to be done with certain scheduling requirements
  - Typically periodic tasks

Specifying Operation Scheduling Properties

- *Problem*
  - Different operation have different scheduling requirements
  - Operation scheduling properties must be complete
    - The system-wide scheduling policy has specific data requirements in order to guarantee schedulability
  - Operation scheduling properties must be abstract
    - Scheduling policies and mechanisms can change as the project evolves

Specifying Operation Scheduling Properties

- *Solution*
  - Define an `RT_Operation` interface
    - Must be implemented by all object with scheduling requirements
    - Allows `RT_Operation`s to share scheduling properties (*e.g.*, period, priority, etc) with between operations and other Execution Model API's
  - `RT_Operation` is integrated into ACE
    - Portable to Win32, Solaris, POSIX 1003.1c, VxWorks, etc.

The RT_Operation Interface

- If objects encapsulate operations with scheduling requirements, then object methods are the entry points of execution

- Each `RT_Operation` contains an `RT_Info` descriptor:

```
struct RT_Info
{
  Time worst_case_execution_time;
  Time typical_execution_time;
  Time cached_execution_time;
  Period period;
  Priority priority;
  Time quantum;
  sequence <RT_Info> called_tasks;
  // ...
};
```

Using RT_Operation

- A class that implements `RT_Operation` defines an `RT_Info` descriptor for each method.

- `Scheduled_Method` describes the execution properties of a single method
  - Execution time → worst case and average case method execution times
  - Period → the rate the method executes
  - Quantum → max time to run before preempting for same priority tasks
  - Priority → allows "clients" to assign levels of importance
    - Not applicable for Rate Monotonic Scheduling

Advantages to RT_Operation API's

- Scheduling mechanisms acquire operation scheduling properties via `RT_Info` interfaces
  - Event Channels make scheduling decisions based on data from Suppliers and Consumers

- Abstract interfaces support changes in scheduling policy

- Facilitates simulation-time logging of scheduling data
  - Off-line proof of schedulability
  - Integration with 3rd party scheduling utilities

Event Channel Scheduling Mechanisms

- *Problem*
  - Event Channels must implement system-wide scheduling policies during event propogation
- *Solution*
  - Channels use `RT_Operation` and `RT_Info` interfaces to obtain task scheduling properties
  - Channels can utilize multiple concurrency strategies to implement scheduling policies

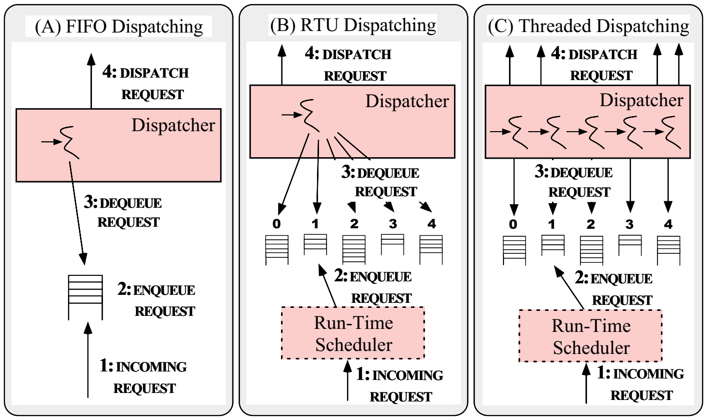
---

Concurrency Strategies

- *Problems*
  - The system-wide scheduling policy may require that Event Channels delegate threads to Suppliers and Consumers
    - Real-time threads can guarantee that higher rate tasks preempt lower rate task in a Rate-Monotonically scheduled system
- *Solution*
  - Event Channel push and pull operations can be entry points for channel-maintained threads
  - A channel's concurrency policy can be decided by a global scheduling component

---

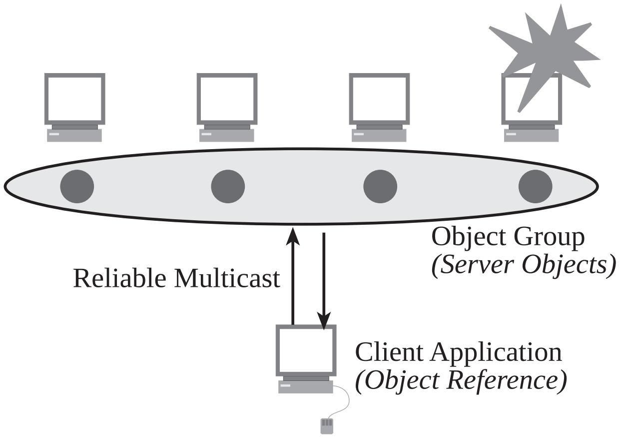
Concurrency Alternatives



Related Patterns and Architectures

- *Observer* (Gamma, Helm, Johnson, Vlissides)
  - "Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically."
- *Publisher-Subscriber* (Buschmann, Meunier, Rohnert, Sommerlad, Stal)
  - "Helps to keep the state of cooperating components synchronized. To achieve this, it enables one-waypropagation of changes: one publisher notifies any number of subscribers about changes to its state."
- *Object Group* (Silvano Maffeis)
  - "Provides a local surrogate for a group of objects distributed across networked machines."

Overview of Object Group Architecture



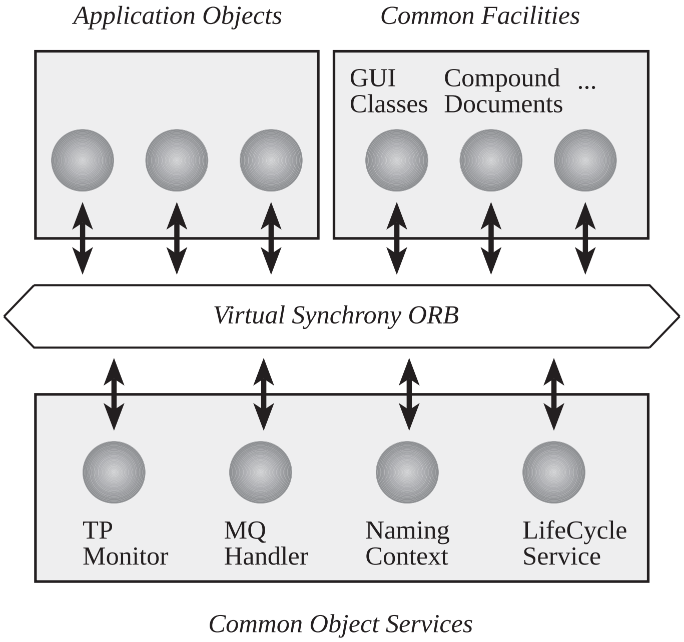
Object Group
(Server Objects)

Reliable Multicast

Client Application
(Object Reference)

- Based on "Virtual Synchrony"
  - http://www.olsen.ch/~maffeis/

Electra Overview



Application Objects

Common Facilities

GUI Classes

Compound Documents

...

Virtual Synchrony ORB

TP Monitor

MQ Handler

Naming Context

LifeCycle Service

Common Object Services

Summary

- The OMG event services specification defines a decoupled communication model between distributed objects
  - This model enables asynchronous communication between *suppliers* and *consumers*
- The OMG event services specification is useful for devising the basis for a flexible "publish/subscribe" service
- Implementations are slowly coming on line
  - Main problem is lack of standard semantics...
- RT Event Service integrated with TAO
  - www.cs.wustl.edu/~schmidt/TAO-obtain.html