Asynchronous RPC
Abstraction

- **Motivation**
  - Strong coupling induces strong dependencies in distributed application
  - Latency of remote invocations different than local ones
  - Remove flow coupling

- **(Subtypes of) proxy abstraction**
  - *Oneway* call: no return value
  - *Explicit* future: no inherent waiting for return value
  - *Implicit* future: no necessary waiting for return value

- **From concurrent programming**
  - Usually based on specific compilation
Oneway Call

- **Abstraction**
  - Proxy

- **Type**
  - Same as server object
  - Reply-less calls can have different semantics by default, or
  - Methods without return value can be marked, e.g, `oneway`
    ```java
    public interface Bob {
        public void helloBob();
        public String howIsBob();
    }
    ```
Remote invocation returns without waiting

✓ For termination of the remote method body execution
✓ Or even without awaiting an ack for successful reception

Obviously flow coupling is removed

✓ When anyway not necessary, or by reasoning/designing differently, e.g.,
✓ 1 synchronous invocation -> 2 asynchronous invocations

```java
public interface BobCallback {
    oneway public void BobIs(String s);
}

public interface Bob {
    oneway public void howIsBob(BobCallback bcb);
}
```
E.g.,
- ABCL, Actalk, Hybrid, PO, CORBA
- With or without different semantics for reply-less methods

Anthropomorphism
- A hologram which can not talk, only listen
- Transmits what initiator says
- Replies (if any) must be sent by some other means, e.g., a second hologram which only talks
Explicit Future - Polling

- **Principle**
  - Return *future* object instead of return value
  - Return value can be queried through *future* object

- **Abstraction**
  - *Future proxy*

- **Type**
  - Original server object type modified/subtyped
  - Maybe use of keyword to mark methods, e.g., *future*

- **E.g.,**
  - ABCL, Act++, CSmalltalk, PO, CORBA AMI
Each return value type $T$

- Is changed to $T\text{Future}$, e.g.,

```java
public interface Bob {
    public String howIsBob();
}
```

- Becomes

```java
public interface BobProxy extends Bob, Proxy {
    public String howIsBob();
    public StringFuture howIsBob();
}
```
Basic future type

```java
public abstract class Future {
    public boolean isAvailable() {...}
    public void wait() {...}
    ...
}
```

Hence StringFuture is something like

```java
public class StringFuture extends Future {
    public String poll() {...}
    public String pull() {...}
    ...
}
```
Example

- **Consumer**
  - ✓Client
  - ✓Invoker

  ```java
  Bob bob = new BobProxyImpl(...);
  StringFuture how = bob.howIsBob();
  System.out.println(how.pull());
  ```

- **Producer**
  - ✓Server
  - ✓Invokee

  ```java
  public class BobServer implements BobSkeleton {...}
  Bob bob = new BobServer(...);
  bob.howIsBob();
  ```
Explicit Future - Callback

- When polling for replies with futures
  - Flow coupling is reduced, but not entirely avoided
  - Only callback mechanism can fully remove flow coupling

- Abstraction
  - Callback proxy

- Type
  - Original server object type Is modified/subtyped
  - Maybe use of keyword to mark methods, e.g., future

- E.g.,
  - ABCL, Act++, CSmalltalk, CORBA AMI
Each return value type $T$

- Is changed to `void`, and the operation gets an additional argument, e.g.,

```java
public interface Bob {
    public String howIsBob();
}
```

- Becomes (generated by compiler)

```java
public interface BobProxy extends Bob, Proxy {
    public String howIsBob();
    public void howIsBob(StringCallback c);
}
```
Basic callback type

```java
public interface Callback {
    public void exceptionOcc(Exception ex);
    ...
}
```

Hence StringCallback is something like

```java
public interface StringCallback extends Callback {
    public void reply(String s);
}
```

Implemented by application
Example

- **Consumer**
  - Client
  - Invoker

```java
public class mySCB implements StringCallback {
    public void reply(String s) {
        System.out.println(s);
    }
    ...
}
Bob bob = new BobProxyImpl(...);
bob.howIsBob(new mySCB());
```

- **Producer**
  - Server
  - Invokee

```java
public class BobServer implements BobSkeleton {
    ...
    Bob bob = new BobServer(...);
    bob.howIsBob();
bob.howIsBob(new mySCB());
    > good
```
Implicit Future

Principle

- Developer
  - Use the return value of a remote invocation as late as possible in the code

- System
  - Return immediately from the call
  - Even though the value is not fixed
  - "Insert" it as soon as available
  - If it is queried before, block
Abstraction

✓ Proxy

Type

✓ Same as original type
✓ Future by default, or explicitly mark future calls e.g., `future`

```java
public interface Bob {
    public String howIsBob();
}
```

E.g.,

✓ Eiffel//, Karos, Meld
Illustration

Bob bob = ...; /* lookup */
/* remote call is issued */
String how = bob.howIsBob();
...
/* how might still be undefined */
...
/* if how is still undef., this will block */
System.out.println(how);
...
Anthropomorphisms

- Explicit future - polling
  - A hologram which does (can) not answer to your question
  - Instead you get a phone nb etc., where you can get the reply
  - You might only get « occupied » if the reply is not ready
  - E.g., hologram of a stupid and lazy assistant who promises to find the answer for you, or tells you where to find it

- Explicit future – callback
  - A hologram which does not reply immediately, still « listens »
  - When a reply is ready, you will be called
  - E.g., hologram of a stupid but nice assistant who will find out some details calls you
Implicit future

- A hologram which replies immediately with some « superficial » reply
- You think you have understood, yet you have not
- If you need more details immediately, you must give the hologram time
- Otherwise, you can continue, the hologram will give you more information later anyway, which will help understand - « aha » effect
- E.g., hologram of a smart assistant: you ask a question, and she/he does not know the reply, so you are drowned in superficial talk giving him/her more time to find the right reply
Asynchronous RPC

CORBA Asynchronous Messaging Interface (AMI)
Asynchr. Invocations in CORBA

- CORBA enables
  - Oneway operations with best-effort semantics
  - Explicit future with DII
  - And offers the *Asynchronous Messaging Interface (AMI)*:

- Consisting of two parts
  - Interface for specifying invocation policies (qualities of service for invocations)
    - *Implied IDL*: asynchronous variants of IDL defined operations
      - Future
      - Callback
For each interface <I>, the IDL compiler generates

✓ An AMI_<I>Poller value type for futures
✓ An AMI_<I>Handler interface for callbacks
✓ An AMI_<I>ExceptionHolder value type for exceptions

Example

interface Agenda {
    void add_appointment(in Time at, in Person p)
        raises (AlreadyTaken);
    void remove_appointment(in Time at)
        raises (NoAppointment);
    Person get_appointment(in Time at);
};
interface Agenda
{
    AMI_AgendaPoller sendp_add_appointment(in Time at,
                                            in Person p);
    AMI_AgendaPoller sendp_remove_appointment(in Time at);
    AMI_AgendaPoller sendp_get_appointment(in Time at);

    void sendc_add_appointment(in AMI_AgendaHandler handler,
                                in Time at, in Person p);
    void sendc_remove_appointment(in AMI_AgendaHandler handler,
                                  in Time at);
    void sendc_get_appointment(in AMI_AgendaHandler handler,
                               in Time at);
};
interface AMI_AgendaPoller : Messaging::Poller
{
    void add_appointment(in unsigned long timeout) 
        raises(AlreadyTaken, CORBA::WrongTransaction);
    void remove_appointment(in unsigned long timeout) 
        raises(ToAppointment, CORBA::WrongTransaction);
    void get_appointment(in unsigned long to, out Person p); 
};

- Only in/inout parameters remain in operation
  - inout parameters are changed to in

- out/inout parameters including any return value are added to corresponding parameters of poller operation
  - inout parameters are transformed to out
Handler

interface AMI_AgendaHandler : Messaging::ReplyHandler
{
    void add_appointment();
    void add_appointment_excep(in AMI_AgendaExceptionHolder h);
    void remove_appointment();
    void remove_appointment_excep(in AMI_AgendaExceptionHolder h);
    void get_appointment(in Person p);
    void get_appointment_excep(in AMI_AgendaExceptionHolder h);
};

☐ xxx_excep() method is called if an exception occurred
value AMI_AgendaExceptionHolder : Messaging::ExceptionHolder
{
    void raise_add_appointment()
        raises (AlreadyTaken);
    void raise_remove_appointment()
        raises (NoAppointment);
    void raise_get_appointment();
};

- Additional information, e.g., target can be obtained through supertypes of
  - ExceptionHolder, ReplyHandler, Poller
Publish/Subscribe
Origins

- **Group-based systems**
  - Inherently one-to-n

- **Anonymous Communication**
  - Derived from Generative Communication (Tuple Space)
  - Targeting at strong decoupling of participants, e.g., for
  - Very large, long-lasting, scalable applications
  - Based on *Information Bus* abstraction, several flavors
    - *Topic*-based: iBus, SmartSockets, TIBCO, Vitria / JMS, CORBA Event & Notification Srvcs, some with a touch of
    - *Content*-based: Gryphon, Siena, Jedi, JavaSpaces, DACs
    - *Type*-based: GDACs, JavaPS
Publish/Subscribe

- **Model**
  - Producers *publish* information
  - Consumers *subscribe* to information
  - Usually producers and consumers both in *push* mode

- **Decoupling of participants**
  - In time
  - In space
  - In flow

- **Enforces scalability**
Abstraction

- Channel

Basic type

```java
public interface Channel {}
```

- With `publish()` and `subscribe()` methods to publish/subscribe to events
- Signatures varying according to flavor
Abstraction

✓ Untyped channel

Type

```java
public interface UntypedChannel {
    void publish(Event e);
}
```

✓ Events are usually « serializable » data objects

```java
public interface Event extends Serializable {
}
```

✓ In most cases, only predefined types are allowed, e.g., self-describing messages, including QoS description
Topic-Based Publish/Subscribe

- A.k.a. subject-based publish/subscribe

- News-like approach
  - Messages are classified according to topic names, e.g., EPFL
  - Topics can be seen as (dynamic) groups

- URL-like topic names for convenience
  - Topics arranged in a hierarchy, e.g., /EPFL/DSC
  - Automatic subscriptions to subtopics
  - Wildcards
  - Aliases
Topic-Based
Abstraction

✓ Topic

Type

```java
public interface Topic extends UntypedChannel {
    public String getTopicName();
    public void subscribe(Notifiable n);
    public void unsubscribe(Notifiable n);
}
```

Where a Notifiable represents a callback object

```java
public interface Notifiable {
    public void notify(Event e);
}
```

One-to-n, all-of-n
Example

```java
public TopicImpl implements Topic {
    public TopicImpl(String topicName) {
        ...
    }
}
public class State implements Event {
    public String mood;
    public String name;
    public State(String m, String n)
    {
        mood = m;
        name = n;
    }
}

Producer (publisher)

Topic t = new TopicImpl("/People/Moods");
Entry bobState = new State("good", "Bob");
t.publish(bobState);
public class StateNotifiable implements Notifiable {
    public void notify(Event e) {
        if (e instanceof State) {
            State s = (State)e;
            System.out.println(s.name + "is doing" + s.state);
        }
    }
}

.Consumer (subscriber)
Topic t = new TopicImpl("/People/Moods");
t.subscribe(new StateNotifiable());

> Bob is doing good
> Alice is doing better
Content-Based Publish/Subscribe

- A.k.a. property-based publish/subscribe
- Events classified according to their properties
  - Consumers subscribe by specifying properties of events of interest
  - Application criteria are seen as subscription pattern
  - Translated to filter, or predicate, matched against events

- Classic approach
  - Map event attributes to properties
  - Subscription language and parser,
  - E.g., "name == ‘Bob’"
Content-Based
Abstraction

Type

public interface ContentChannel extends UntypedChannel {
    public void subscribe(Notifiable n, Filter f);
    public void unsubscribe(Notifiable n);
}

Where a Filter represents the subscriber’s individual criteria

public interface Filter {
    public boolean conforms(Event e);
}

One-to-n, some-of-n

Attention: filtering opaque!
Self-Describing Messages

Similar to DynAny in CORBA

- Represent rather *structures* than objects, e.g.,

```java
public class SelfDescribingEvent extends Event {
    public void addString(String fieldName, String s) {...}
    public void addByte(String fieldName, Byte b) {...}
    public void addObject(String fieldName, Object o) {...}
    ...
    public String getString(String fieldName) {...}
    public Byte getByte(String fieldName) {...}
    public Object getObject(String fieldName) {...}
    ...
    public String[] getFieldNames() {...}
    public Class getFieldType(String fieldName) {...}
    ...
}
```
public class Contents {
    public static ContentChannel getChannel() {...} ... }

- **Producer (publisher)**

    ContentChannel channel = Contents.getChannel();
    Event bobState = new SelfDescribingEvent();
    event.addString("name", "Bob");
    event.addString("mood", "good");
    event.addInteger("time", 15);
    channel.publish(bobState);
public class ContentNotifiable implements Notifiable {
    public notify(Event e) {
        if (e instanceof SelfDescribingEvent) {
            SelfDescribingEvent s = (SelfDescribingEvent)e;
            System.out.print(s.getString("name") + " feels ");
            System.out.print(s.getString("mood") + " at ");
            System.out.println(s.time);
        }
    }
}

☐ Consumer (subscriber)

ContentChannel content = Contents.getChannel();
Filter filter = new AttributeFilter("time > 21 && (mood == ‘good’ || mood == ‘very good’)");
content.subscribe(new StateNotifiable(), filter);
Most topic-based systems nowadays also incorporate content-based features

- More flexible
  - Can be used to express topics

Self-describing messages

- Offer much dynamism
- Enforce interoperability
- Rarely required
- Not «type-» safe
Oneway Proxy Approach

- Reuse proxy abstraction
  - Without replies
  - Dispatch request through channel, to all consumers of same type as proxy

- Abstraction
  - Proxy (oneway)

- Type
  - Consumer type
    ```java
    public interface Bob {
      oneway public void helloBob();
    }
    
    public interface BobProxy extends Bob, Proxy {}```
Example

- **Producer**
  - `Publisher`
  ```java
  public class BobProxyImpl extends ProxyImpl implements BobProxy {
    public BobProxyImpl(
      ContentChannel c)
    { super(c); } ... }
  ContentChannel content = Contents.getChannel();
  Bob bob =
    new BobProxyImpl(content);
  bob.helloBob();
  ```

- **Consumer (i-th)**
  - `Subscriber`
  ```java
  public class BobServer extends BobSkeleton {
    public BobServer(
      ContentChannel c)
    { super(c); } ... }
  ContentChannel content = Contents.getChannel();
  Bob bobi = new BobServer(content);
  bobi.helloBob();
  ```
- E.g.,
  - COM+: based on application-provided « dummy » proxy
    - Content-based: values for invocation arguments
  - CORBA Event Service: precompiler based

- Note
  - Also been applied to synchronous « group » invocations
  - E.g., transform all return types to array types, and return after obtention of a certain number of replies
  - Similarly, futures and callbacks can be used to return replies
  - Coupling?
Genericity Approach

- **Abstraction**
  - *Typed channel (type-parameterized channel)*

- **Type**

  ```java
  public interface TypedChannel<EventType> {
    void publish(EventType e);
    void subscribe(TypedNotifiable<EventType> n,
                    TypedFilter<EventType>);
  }
  
  With
  public interface TypedNotifiable<EventType> { public void notify(EventType e); } 
  public interface TypedFilter<EventType> { public boolean conforms(EventType e); }
  ```
Based on parametric polymorphism

- A form of genericity
- Channels are type parameterized by the effective event type
- Bounded parametric polymorphism is used to ensure « serializability »:
  ```java
  public interface TypedChannel<EventType extends Event> {}
  ```
- The application can use its own event (sub)types
Type-Based Publish/Subscribe

- **Subscription criterion**
  - The *type* (its interface) of application-defined events
  - Content-based queries based on *methods*

- **Combines static and dynamic schemes**
  - Static classification should be made as far as possible for *efficiency*
  - Filters for fine-grained content-based subscription increase *expressiveness* if required

- **Languages which support *structural reflection***
  - No need for specific events (e.g., Java *introspection*),
  - In other languages, events can *subtype* an introspective event type
Type-Based
**Example**

```java
public class TypedImpl<T> implements TypedChannel<T> { ... } 

public class State implements Event {
    private String mood;
    private String name;
    private long time;
    public String getMood() { return mood; }
    public String getName() { return name; }
    public long getTime() { return time; }
    public State(String m, String n, long t) {
        mood = m;
        name = n;
        time = t;
    }
}

- **Producer side**

  TypedChannel<State> stateChannel = new TypedImpl<State>();
  State bobState = new State("good", "Bob", 15);
  stateChannel.publish(bobState);
```
public class StateNotif implements Notifiable<State> {
    public notify(State s) {
        System.out.print(s.getName() + " feels ");
        System.out.print(s.getMood() + " at ");
        System.out.println(s.getTime());
    }
}

Consumer side

TypedChannel<State> stateChannel = new TypedImpl<State>();
Subscription sub = stateChannel.subscribe(new StateNotifiable());
sub.constrain().getMood().equals("good");
sub.start();
Two primitives added, e.g.,

```java
class State {
    String name;
    String mood;

    State(String name, String mood) {
        this.name = name;
        this.mood = mood;
    }

    public String getName() {
        return name;
    }

    public String getMood() {
        return mood;
    }
}
```

```java
publish new State(...);
subscribe (State s)
    { return s.getMood().equals("good"); }
    { System.out.println(s.getName()); };
```

Heterogenous transl. with extended compiler

Generation of “adapter” code, e.g., StateAdapter, invocations of primitives transformed

Abstract syntax trees for content filters
Publish/Subscribe

Java Message Service (JMS)
Java Message Service

- **The Java Message Service is only an API**
  - Standardized API for messaging in Java
  - Implemented by most industrial solutions
    - TIBCO
    - iBus
    - Gryphon
    - ...

- **Two messaging styles:**
  - Publish/subscribe (topic-based & content-based): some-of-n
  - Point-to-point (message queuing): one-of-n
Benefit of JMS

- **Sun standard**
  - Ensures a certain degree of portability
  - Integration with other Java concepts/services
    - *Enterprise Java Beans* (EJB): asynchronous beans vs. synchronous beans
    - *Java Database Connectivity* (JDBC) for database integration
    - *Java Transaction Service* (JTS) for messages as part of distributed transactions
    - *Java Naming and Directory Intf* (JNDI) for object lookup
  - API can be downloaded: package `javax.jms`
JMS Event Model

- General-purpose messages which require explicit marshalling

- Message body can contain
  - Stream
  - Properties
  - String
  - Object
  - Bytes

- Additional attributes
  - Message header: explicit messaging
  - Message properties: for content-based filtering
Message Attributes

- **Message header**
  - Assigned by service upon send
    - Destination
    - Delivery mode (PERSISTENT, NON_PERSISTENT)
    - Message ID
    - Timestamp
    - Priority
    - Expiration
  - Provided by client
    - Correlation ID, e.g., refer to other message
    - Type
    - Reply destination

- **Message properties**
  - Name-to-value properties provided by message producer
  - Property types (native Java types)
    - boolean
    - byte
    - short
    - int
    - long
    - float
    - double
    - String
  - Note: attributes mapped to properties, encapsulation...!
Properties for Content-Based

- Properties of messages are assigned explicitly
  - Not java.util.Properties
- Subscriber describes required properties
  - Message selector = filter
  - Subscription language: message selector is String
  - Syntax specified by JMS
  - Must be mapped to service provider’s subscription language syntax
- E.g.,
  "JMSType = 'car' AND color = 'blue' AND weight > 2500"
Common Facilities

- **Destination**
  - Named object (topic, queue) obtained through JNDI: empty interface

- **ConnectionFactory**
  - Obtained through JNDI, used to create `Connection` to a topic, queue: empty

- **Connection**
  - May require authentication
  - Register `ExceptionListener` for problem detection
  - Factory for `Session`

- **Session**
  - Required by client (producer/consumer) to interact with topic, queue
  - Creates `MessageProducer (push)`, `MessageConsumer (push/pull)`
  - Single threaded. Transaction support, unacknowledged messages, order ...

public interface Connection {

    public String getClientID() throws JMSException;
    public void setClientID(String ID) throws ...

    public void setExceptionListener(ExceptionListener l) throws ...
    public ExceptionListener getExceptionListener() throws ...

    public void close() throws ...
    public start() throws ...
    public stop() throws ...

    /* (Sessions created through implementation classes) */
}
public interface Session {

    public void setMessageListener(MessageListener l) throws ...
    public MessageListener getMessageListener() throws ...

    public TextMessage createTextMessage() throws ...
    public StreamMessage createStreamMessage() throws ...

    ...

    public void close() throws ...
    public void recover() throws ...
    public void commit() throws ...
    public void rollback() throws ...

    ...
}
public interface MessageProducer {

    public void setDeliveryMode(int deliveryMode) throws ...;
    public int getDeliveryMode() throws ...;

    public void setPriority(int defaultPriority) throws ...;
    public int getPriority() throws ...;

    public void setTimeToLive(long ttl) throws ...;
    public long getTimeToLive() throws ...;

    ...
}

public interface MessageConsumer {

    /* Provide content-based filter */
    public String getMessageSelector() throws ...

    /* Push model */
    public void setMessageListener(MessageListener l) throws ...
    public MessageListener getMessageListener() throws ...

    /* Poll */
    public Message receive() throws ...
    /* Blocking pull */
    public Message receive(long timeout) throws ...

}
Point-To-Point (PTP)

- **Objects**
  - Queue represents a vendor-specific implementation
  - TemporaryQueue is a temporary incarnation, bound to a QueueConnection
  - Created through a QueueConnectionFactory
  - QueueSession, QueueReceiver (message consumer: push/pull), QueueSender (message producer)
  - QueueBrowser to query queue without removing messages

... 

- **Note**
  - Message selector can be specified by consumer
Queue

public interface Queue {

    public String getQueueName() throws ...
    public String toString() throws ...
}

public interface QueueBrowser {

    public Enumeration getEnumeration() throws ...
    public String getMessageSelector() throws ...

    public String getQueue() throws ...
    ...
}

Publish/Subscribe

- **Objects**
  - Topic gives access to pub/sub system: no naming conventions
  - TemporaryTopic, TopicConnectionFactory, TopicConnection, TopicSession, as seen previously
  - TopicSubscriber (message consumer) and TopicPublisher (producer)

- **Durable subscription**
  - Client provides unique ID

- **TopicRequestor**
  - Use pub/sub to make request/replies

- **Mixed topic/content-based**
  - Client provides a message selector
public interface Topic {

    public String getTopicName() throws ...
    public String toString() throws ...
}

public class TopicRequestor {

    public TopicRequestor(TopicSession session, Topic topic) 
        throws ... {...}
    public Message request(Message message) throws ... {...}
    ...
}
JMS Exceptions

- **JMSException**
  - Checked exception
  - Root of exception hierarchy

- **Specific exceptions**
  - **JMSSecurityException**: authentication problem
  - **InvalidDestination**: destination not understood by provider
  - **InvalidSelectorException**: «syntax error» in filter
  - **MessageFormatException**: e.g., unsupported payload class

...
Publish/Subscribe

CORBA Event & Notification Services
Overview

- **Untyped events**
  - The effective events are of type `org.omg.CORBA.Any`

- **Typed events**
  - Oneway proxy

- **Structured events**
  - Starting from Notification Service
  - Semi-typed events
  - Set of predefined self-describing messages
Event Service

- **Event Channel abstraction**
  - Event channel is CORBA object
  - Subscriber is called Consumer
  - Publisher is called Supplier
- **Event channel appears as**
  - Supplier for consumers
  - Consumer for suppliers

- **Different interaction styles**
  - Between consumers and channel
  - Suppliers and channel

- **Guarantees**
  - Not necessarily FIFO
  - Best-effort
  - QoS only with Notification Service
Styles

- **Usual:** *push* model

- **Special:** *pull* model
module CosEventComm {
exception Disconnected { }
...
interface PushConsumer {
    void push(in any data) raises (Disconnected);
    void disconnect_push_consumer(); }
interface PushSupplier {
    void disconnect_push_supplier(); }
interface PullConsumer {
    void disconnect_consumer(); }
interface PullSupplier {
    any pull() raises (Disconnected);
    any try_pull(out boolean has_event) raises (Disconnected);
    void disconnect_pull_supplier(); }
};
Setup

- **Channel creation (vendor-specific)**
  - Usually manually, suppliers and consumers connect
  - Certain implementations provide *in-process* channels

- **Supplier connection example**
  1. Get a *supplier admin* for the channel
  2. Get a *proxy push / pull consumer* (subtype of push / pull consumer) from the supplier admin
  3. Connect to proxy consumer

- **Consumer connection example**
  1. Get a *consumer admin* for the channel
  2. Get a *proxy push / pull supplier* (subtype of push / pull supplier) from the consumer admin
  3. Connect to proxy supplier
module CosEventChannelAdmin { }

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(); 
};
Notification Service

- QoS
- New proxies
  - For structured events, e.g., `StructuredProxyPushConsumer`
  - For sequences of structured events, e.g., `SequenceProxyPullSupplier`
- Filtering
- And many more, e.g.,
  - Event type repository
  - Transactions
QoS

- Represented through `PolicySeq`

```cpp
module CosNotification {
    typedef string PolicyName;
    typedef any PolicyValue;
    struct Policy {
        PolicyName name;
        PolicyValue value;
    };
    typedef sequence<Policy> PolicySeq;
    ...
};
```

- Set at different levels, e.g., channel, admin, proxy
Policies

- **Reliability policy**
  - EventReliability, ConnectionReliability: best-effort or persistent
    - best-effort & best-effort: no reliability
    - best-effort & persistent: automatic reconnect after recovery
    - persistent & persistent: « reliable »

- **Order policy**
  - AnyOrder, FIFOOrder, PriorityOrder, DeadlineOrder

- **Discard policy**

- For sequences: *batch size, pacing interval*
module CosNotification {
    typedef PolicySeq OptionalHeaderFields;
    typedef PolicySeq FilterableEventBody;
    struct EventType {
        string type_name;
        string domain_name;
    };
    struct FixedEventHeader {
        EventType event_type;
        string event_name;
    };
    struct EventHeader {
        FixedEventHeader fixed_header;
        OptionalHeaderField var_fields;
    };
    struct StructuredEvent {
        EventHeader header;
        FilterableEventBody filterable;
        any remainder_of_body;
    };
    ...
};
Filtering

- **Header fields**
  - `EventReliability, Priority, StartTime, StopTime, Timeout`

- **Filterable fields**
  - Similar to JMS, fields as properties
  - Notification Service suggests « standard » structured types

- **Two types of filters**
  - **Filter**: affect event *forwarding* by proxies
  - **MappingFilter**: affect event *handling* with respect to QoS
  - Both queried through `match()`, `match_structured()`, `match_typed()` methods
Filter expression

- Based on Default Filter Constraint Language
- An extension of the Trader Constraint Language (cf. Trading Service)
- E.g.,
  - $type_name == `CommunicationAlarm` and $priority < 2
  - $.value1 > $.value2
  - exists $.howIsBob

- Note
  - $type_name is abbreviation for $header.fixed_header.event_type.type_name
Typed Events

- **Oneway proxy approach**

  ✓ E.g.,

  ```java
  interface Bob {
    void say_hi(in String say);
  };
  ```

  ✓ Yields for instance the following interface for a pull consumer

  ```java
  interface PullBob {
    void say_hi(out String say);
    boolean try_say_hi(out String say);
  };
  ```
Publish/Subscribe

Generic Distributed Asynchronous Collections (GDACs)
Distributed Asynchr. Collections

- **Collections**
  - Intuitive by reusing well-known collection abstraction
  - Collection subtypes enable the expression of
    - Different interaction styles, e.g., DAQueues for queuing
    - Different QoS, e.g., DAList has order, DASet is reliable

- **Distributed**
  - Essentially distributed, unlike JavaSpaces, JGL, ...

- **Asynchronous**
  - Callback mechanism: publish/subscribe
## Characteristics of DACs

<table>
<thead>
<tr>
<th>Collection</th>
<th>DACollection</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Storage order</strong></td>
<td><strong>Delivery order</strong></td>
</tr>
<tr>
<td>✓ Deterministic</td>
<td>✓ Unreliable</td>
</tr>
<tr>
<td>✓ None</td>
<td>✓ Reliable</td>
</tr>
<tr>
<td><strong>Duplicates</strong></td>
<td>✓ Certified</td>
</tr>
<tr>
<td><strong>Insertion order</strong></td>
<td></td>
</tr>
<tr>
<td>✓ Explicit</td>
<td></td>
</tr>
<tr>
<td>✓ Implicit</td>
<td></td>
</tr>
<tr>
<td><strong>Extraction order</strong></td>
<td><strong>Extraction order: pull</strong></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
DAC Framework

- **Java DACs**
  - Extension of java.util collections
  ```java
  public interface DACollection
      extends java.util.Collection {
      public boolean add(Object o);
      public boolean contains(Object o);
      public Object get();
      ...
  }
  ```
DAC Interfaces

- **Callback interface**
  
  ```java
  public interface Notifiable {
      public void notify(Object m, String DACName);
  }
  ```

- **Subscribe (all-of-n)**
  
  - Without subtopics: `contains(Notifiable n);`
  - With subtopics: `containsAll(Notifiable n);`

- **Subscribe (one-of-n)**
  
  - Without subtopics: `remove(Notifiable n);`
  - With subtopics: `removeAll(Notifiable n);`
Generic DACs

- DACs for type-based publish/subscribe
  - Represent collections for a specific element type
    - Published elements are of that type
    - Subscribe: all elements received are of that type
  - Compile-time type checks
  - No (explicit) runtime type conversions

- Obtaining a DAC for type $T$
  - Generate a specific DAC type $T$, e.g., precompiler: $TDAC$
  - Parametric polymorphism (genericity)
- **Callback interface**
  ```java
  public interface GNNotifiable<T> {
    public void notify(T t);
  }
  ```

- **GDAC interface**
  ```java
  public interface GDAC<T> {
    public boolean contains(T t);
    public boolean add(T t);
    public T get();
    public boolean contains(GNNotifiable<T> n, 
                            GCondition<T> c);
    ...
  }
  ```
Programming Example

- Event class

```java
class ChatMsg implements java.io.Serializable {
    private String sender;
    private String text;
    public String getSender() { return sender; }
    public String getText() { return text; }
    public ChatMsg(String sender, String text) {
        this.sender = sender;
        this.text = text;
    }
}
```
« Pure » Type-Based

- Create a local DAC proxy
  
  \[
  \text{GDASet<ChatMsg> myChat = new GDAStrongSet<ChatMsg>();}
  \]

- Insert new objects (publish)
  
  \[
  \text{myChat.add(new ChatMsg("Bob", "Hi from Bob");}
  \]

- Advertise interest in new objects (subscribe)
  
  \[
  \text{public class ChatNot implements GNotifiable<ChatMsg> {}
  \text{    public void notify(ChatMsg m) {}
  \text{      System.out.println(m.getText());}
  \text{    }
  \text{  }}
  \text{myChat.contains(new ChatNot());}
  \]

With Content-Based

- **Pattern**
  - **Accessors**: enable the access of some information, e.g. attribute
  - **Conditions**: express single conditions on events

- **E.g., verify if message sender is "Alice"**
  - **Construct accessor (explicitly)**
    ```java
    Accessor<ChatMsg> getAlice =
    new Invoke<ChatMsg>("/getSender", null);
    ```
  - **Construct pattern**
    ```java
    Condition<ChatMsg> fromAlice =
    new Equals<ChatMsg>(getAlice, "Alice");
    ```
  - **Subscribe**
    ```java
    myChat.contains(new ChatNot(), fromAlice);
    ```