

Architecture of the CORBA Component Model

Event Service

Overview

- Event-oriented communication: an alternative for call-based client-server architecture
- Event Service: simple decoupled communication
- Notification Service: Extension of the Event Service
 - Offers greater flexibility

Events & Notifications

- Event: „something that happens“, „occurrence of some sort“, atomic
- Notification: Information about an event (message)
- Each message has a single distinct source, but potentially many recipients (1:n communications)
- Medium may support n:n communications
- Source of messages does not know consumers
 - Recipients are not explicitly addressed
- Emitting a message is typically non-blocking

Objectives and Applications

- Objectives are decoupling and autonomy
 - In space
 - In time
 - Syntactically
 - Semantically
- Distribution of messages (news ticker)
- Management of Telco networks
- Conferencing systems

Example Applications

- Example Scenario
 - Stock exchange
 - Stocks are traded at different exchanges
 - Decoupled scenario: notations are independent of individual trading decisions
 - Notations are distributed to all registered customers
 - Customers can subscribe to all/certain stocks
 - Provision of
 - Frequency of notification
 - Validity of data (e.g. do not communicate rates older than x minutes)
 - ...

First Approach

```
struct Time
{
    string current_date;
    string current_time;
};
interface StockExchange;
struct StockQuote
{
    string stock_id;
    StockExchange market_place;
    double current_quote;
    Time current_time;
};
```

First Approach

```
interface Subscriber
```

```
{
```

```
    void receive (in ::StockQuote current_quote);
```

```
};
```

```
interface StockExchange
```

```
{
```

```
    void subscribe (in ::Subscriber customer);
```

```
};
```

First Approach

- *StockExchange* Implementation:
 - Manages list of *Subscriber* objects
 - On each rate change, all *Subscriber* objects are notified
- Use communication patterns
 - subscribe/publish
 - Interested parties subscribe to news agency
 - News agency emits news messages
 - Push model
 - News messages are emitted actively by the agency

Approach using CORBA Event Service Interfaces

- CORBA Event Service
 - Interfaces standardized by OMG for event services
 - [ftp.omg.org/pub/docs/formal/00-06-15.pdf](ftp://ftp.omg.org/pub/docs/formal/00-06-15.pdf)
 - [ftp.omg.org/pub/docs/formal/98-10-05.idl](ftp://ftp.omg.org/pub/docs/formal/98-10-05.idl)
 - [ftp.omg.org/pub/docs/formal/98-10-06.idl](ftp://ftp.omg.org/pub/docs/formal/98-10-06.idl)
- Event **consumers** and **supplier**
 - Communication patterns **push** and **pull**
 - Push model: Producer is active
 - Pull model: Consumer is active
- Typed and untyped communication
 - Untyped: messages are communicated using the any type

Consumer and Supplier – Push Model

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

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

Consumer and Supplier – Pull Model

```
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();  
};
```

First Approach using CORBA Event Service (Push Model)

```
interface StockExchange2;
```

```
struct StockQuote2 {
```

```
    // ...
```

```
    StockExchange2 market_place;
```

```
    // ...
```

```
};
```

```
interface Subscriber2 : ::CosEventComm::PushConsumer
```

```
{
```

```
interface StockExchange2 : ::CosEventComm::PushSupplier {
```

```
    void subscribe ( in ::Subscriber2 customer);
```

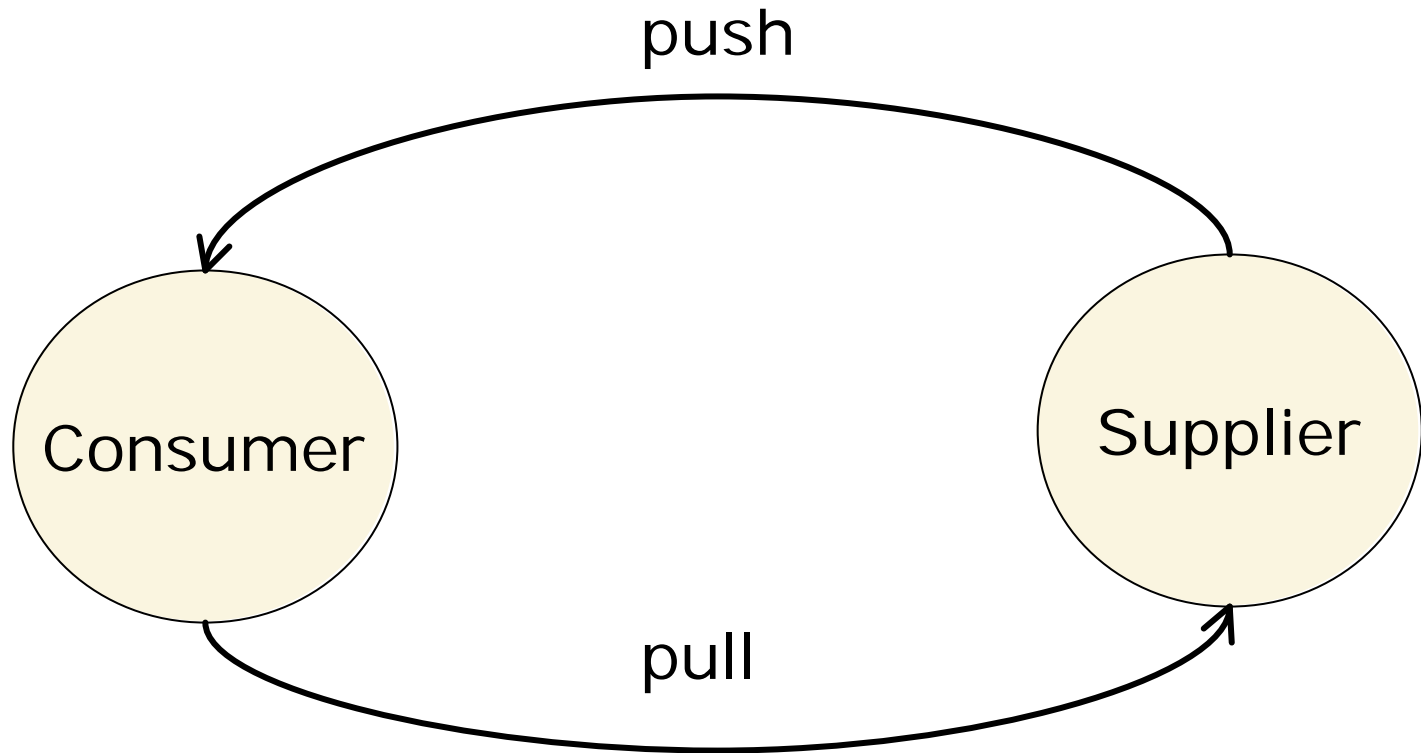
```
};
```

First Approach using CORBA Event Service (Push Model)

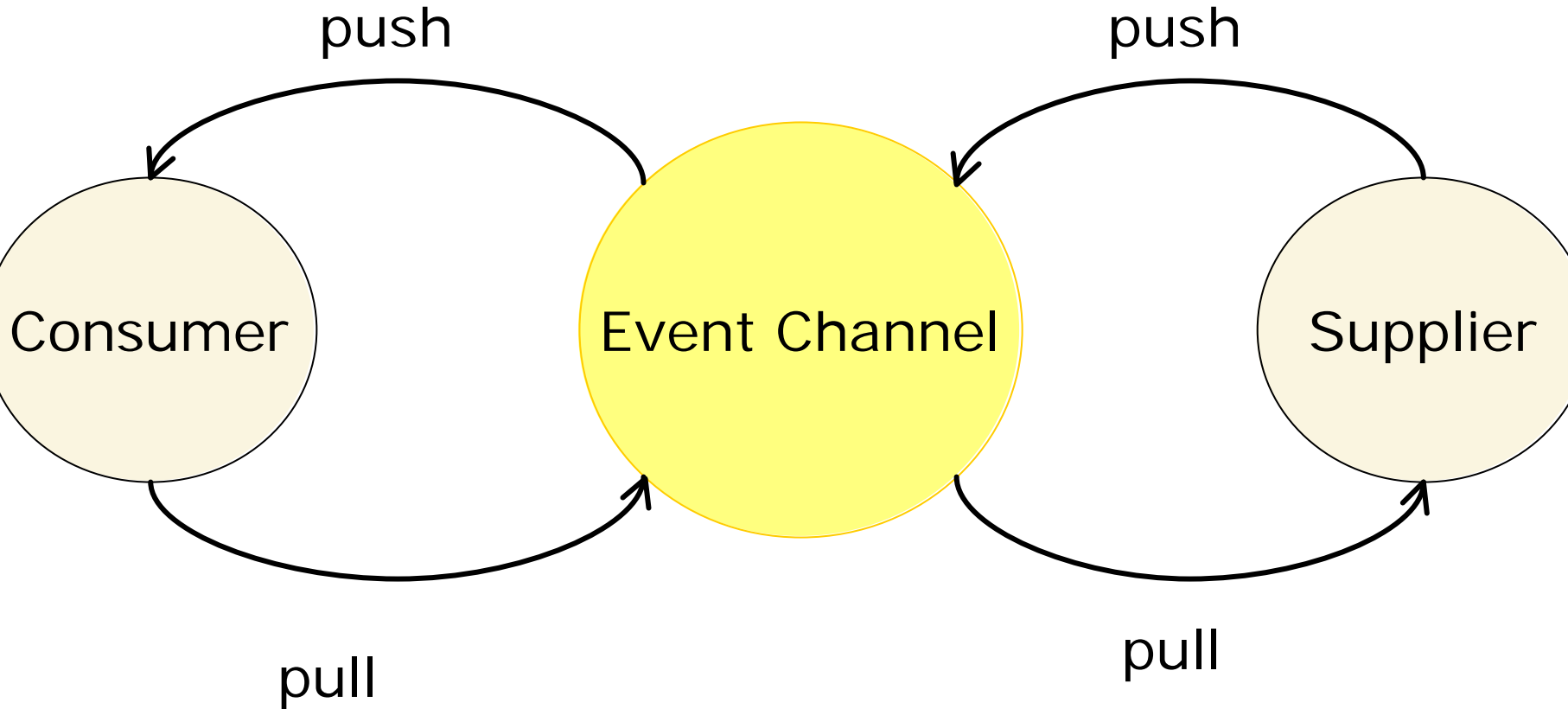
Usage of standard interface for *StockExchange* service

- Push model proves appropriate
- Explicit usage of inheritance from *PushConsumer* and *PushSupplier*
- Problems?
 - *StockExchange* objects are client-aware: all consumers must be known to supplier
 - Solution: Use a middleman between consumer and producer
 - Call the middleman „channel“ and consider the push model

Event Channels



Event Channels



A Channel Definition

```
interface Channel
  : ::CosEventComm::PushSupplier,
  ::CosEventComm::PushConsumer {
void register_supplier (
    in ::CosEventComm::PushSupplier supplier);
void register_consumer (
    in ::CosEventComm::PushConsumer consumer);
};
```

```
interface MyConsumer : ::CosEventComm::PushConsumer
{};
```

```
interface MySupplier : ::CosEventComm::PushSupplier
{};
```


A Channel Definition

- Advantages:
 - Decoupling of communications between *StockExchange* objects and *Subscriber* objects
 - *Subscriber* objects only know the channels they use
 - A single channel can transmit events of multiple suppliers, distributing them transparently to multiple consumers

CORBA Event Channel Interfaces

- Event channel interfaces standardized by OMG
 - Built on top of PushConsumer, PushSupplier, PullConsumer, PullSupplier
 - Usage interfaces (event channel)
 - Management interfaces

Event Channel – Usage Interfaces

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

Event Channel – Usage Interfaces

```
module CosEventChannelAdmin
{
    exception TypeError};

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

Event Channel – Management Interfaces

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

Event Channel – Management Interfaces

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

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

StockExchange Event Service, Using *EventChannels*

- Specification of *Subscriber2* and *StockExchange2* can be reused
- Procedure
 - *StockExchange2* object receive (magically yet) an object reference of a *EventChannelAdmin* object
 - Instantiate via
EventChannelAdmin->for_suppliers()->obtain_push_consumer()
ProxyPushConsumer object, which they use to supply events
 - Registration via **connect_push_supplier()**

StockExchange Event Service, Using *EventChannels*

- Procedure

- *Subscriber2* object receive (magically yet) an object reference of an *EventChannelAdmin* object

- Instantiate via

- EventChannelAdmin->for_consumers()**
->obtain_push_supplier()

- a *ProxyPushSupplier* object, from which they will receive events

- Register there using **connect_push_consumer(...)**

Results so far

- Usage of standardized interfaces for middleman objects of the StockExchange service
 - Simple administration and usage
 - Decoupled communication between supplier and consumer
- Open issues:
 - How to obtain object reference for an EventChannel?
 - Answer:

```
CORBA::Object_var obj = orb->resolve_initial_references("EventService");  
CosEventChannelAdmin::EventChannel::_narrow(obj);
```
 - Problem: this only allows for a single channel shared by all producers and consumers

Further Concepts

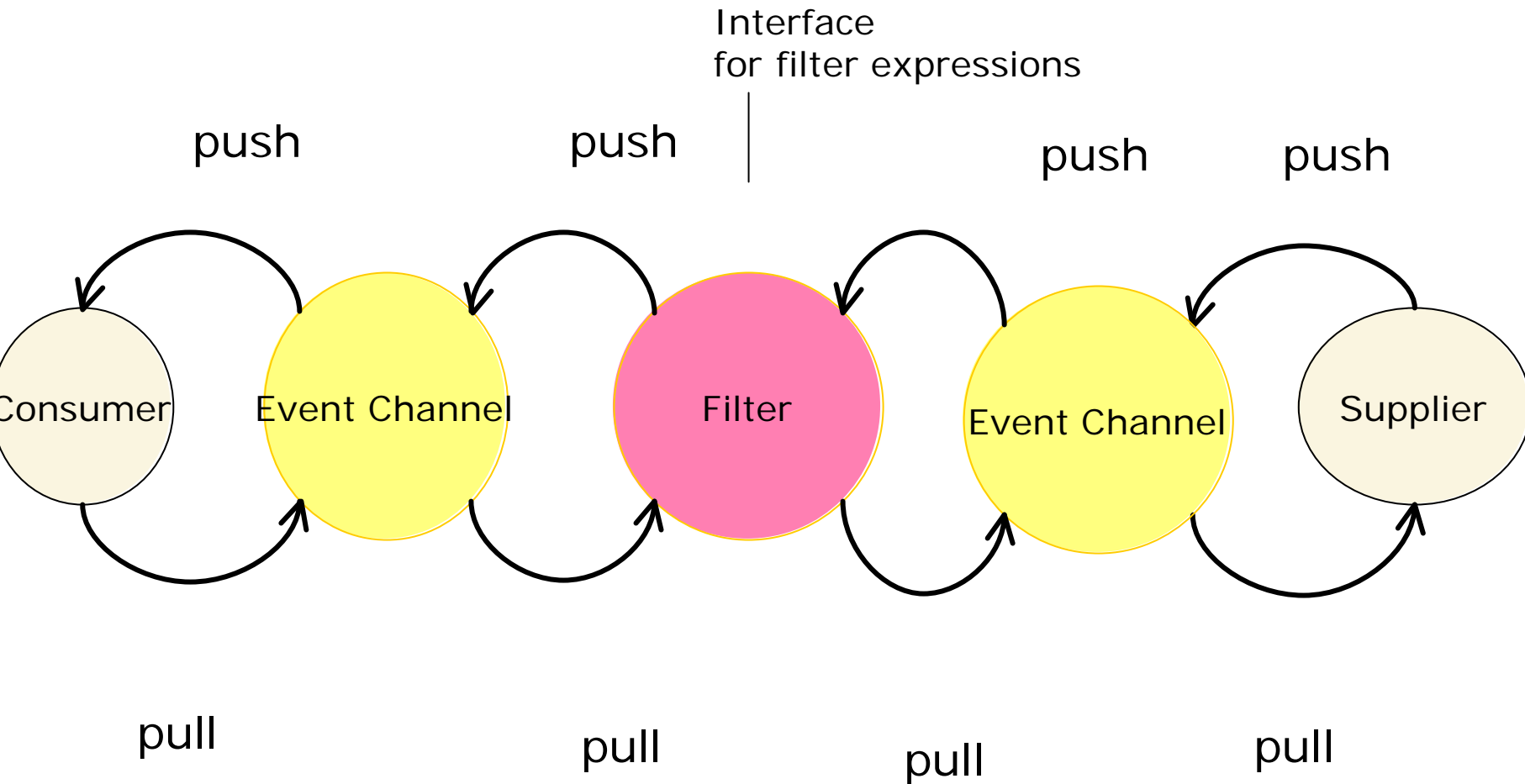
- EventChannel factories
 - Objects able to create EventChannel object:
 - For EventChannels NOT standardized
 - Proprietary solutions (e.g. ORBacus)

```
interface EventChannelFactory {  
    CosEventChannelAdmin::EventChannel  
        create_channel(in ChannelId id);  
    CosEventChannelAdmin::EventChannel  
        get_channel_by_id(in ChannelId id);  
    ChannelIdSeq get_channels();  
    void shutdown();  
};
```

Further Concepts

- Event filters
 - Usable in stock exchange service
 - Subscriber may only be interested in specific stocks
 - Stop-loss: subscriber may be interested only if the rate is below a certain value
 - Stop-bye: subscriber is interested only in rates above a certain value
- Can be implemented as middleman between channels with additional interfaces to install filters

Further Concepts



Further Concepts

- Quality of service for event transmission
 - Event validity/timeout
 - Delay of events
 - Guarantee of delivery
- Canonical extension of the Event Service with these concepts is the Notification Service
 - OMG standard
 - Compatible with Event Service through inheritance