CORBA - IV

Prof. Dave Bakken

Cpt. S 464/564 Lecture

November 13, 2000

Administrative Items

- Handouts
  - New Schedule
  - Event example code (PushModel.C & PushView.C)
- Much of this lecture is from Hennig and Vinoski chapter 20
- New grade breakdown for class:

Component	464	564
Exams (2):	40%	30%
Homework (4) and Surprise Quizzes :	20%	20%
Projects (4):	40%	40%
Participation	0%	10%

Different Interaction Styles

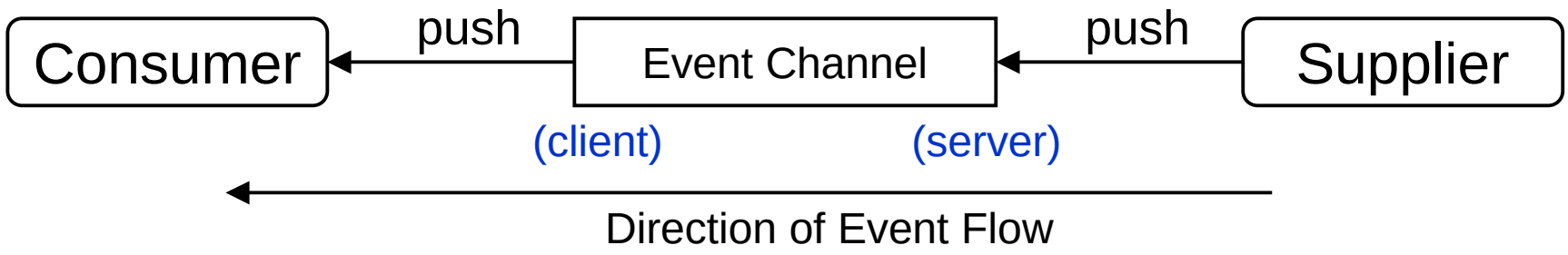
- Synchronous Invocation
  - Client actively invokes requests on passive server
  - Client blocks until reply arrives
  - Note clients are aware of their servers
- This style is too restrictive for some kinds of applications
  - Even asynchronous invocations only help some
- Events
  - Supplier: entity producing the information of interest
  - Consumer: entity receiving and using the information of interest
  - Suppliers can send messages to one or more consumers with a single call
  - Suppliers and Consumers are decoupled: they are not aware of each other's identity

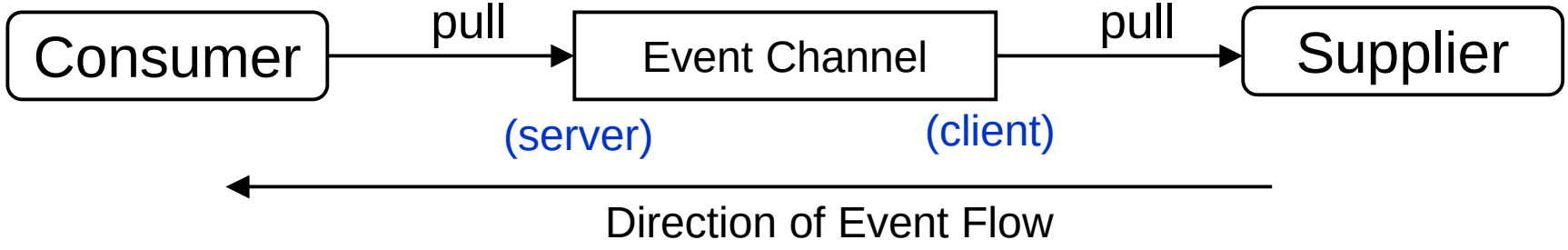
Invocations and Events Contrasted

- Topology
  - Invocations have a single target
  - Events can be delivered to multiple consumers with one call by a supplier
- Coupling
  - Invocations require the client to be aware of the server
  - Events keep the supplier and consumer decoupled, unaware of each other (not referring to each other)
- Blocking
  - Synchronous invocation blocks until invocation returns, so client and server are (loosely) synchronized at some time
  - Events are non-blocking: supplier does not block until all messages have reached all consumers
- Syntactic checking and type safety
  - Invocations are type checked because method's IDL describes all data
  - Events' data needs to be self-describing, generic
    - So types not checked
    - Case 1: consumers know what type of data to expect
    - Case 2: consumers inspect the self-describing type to see type

Event Service Basics

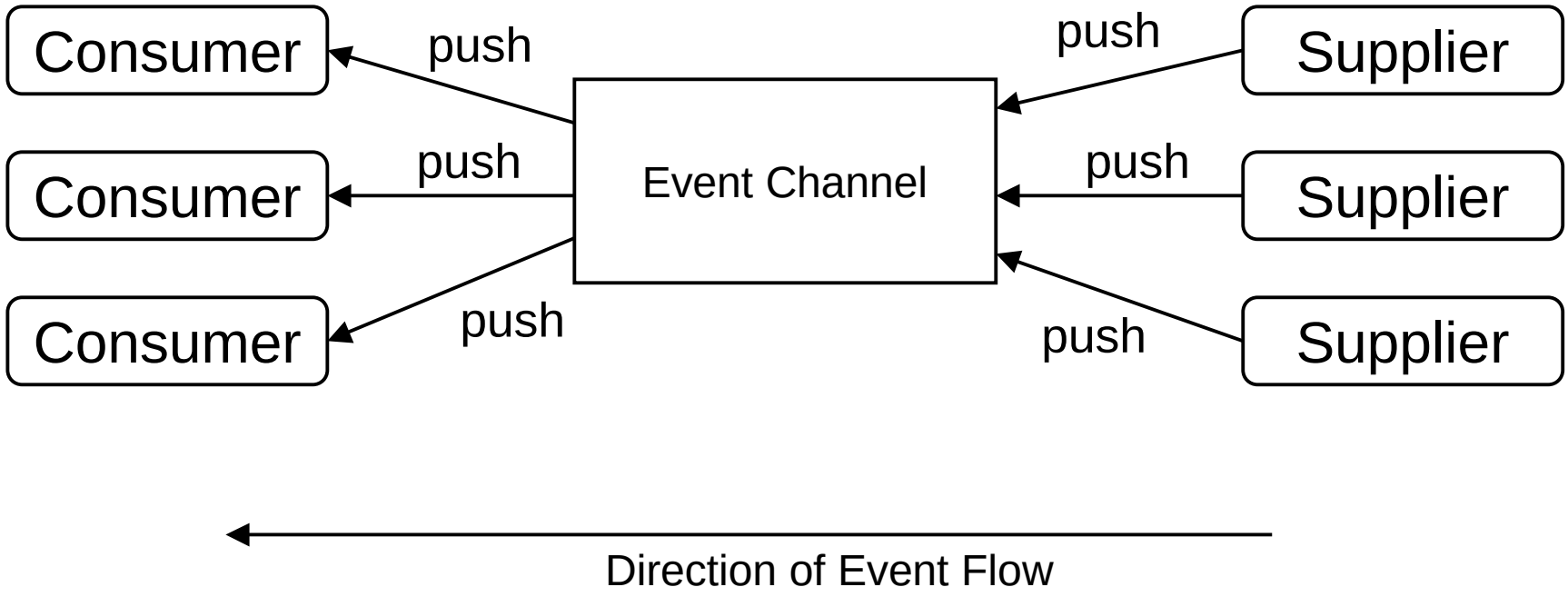
- OMG has Event service
- Both suppliers and consumers connect to an event channel
- Two models: push and pull (4 variants ("models") supported)





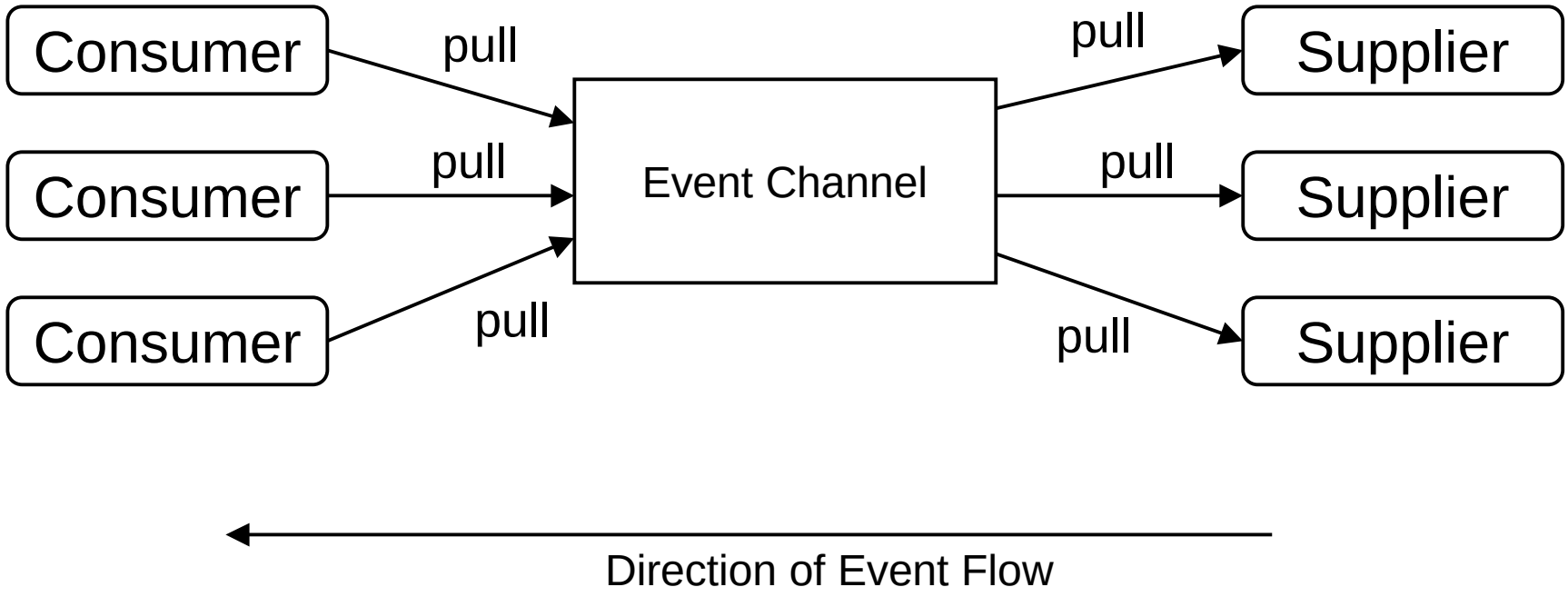
#1: Canonical Push Model

- Suppliers are initiators of events
- Consumers passively wait to receive them
- Event channel plays role of notifier
- Most commonly used event delivery model



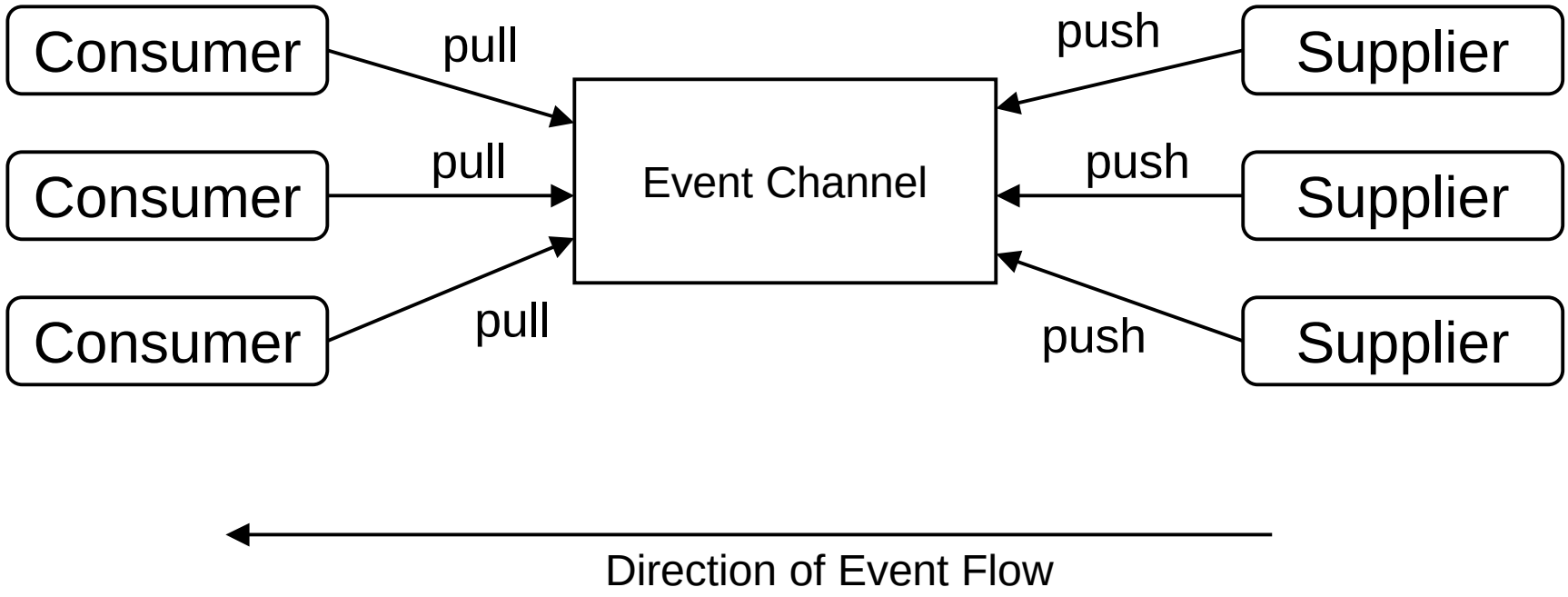
#2: Canonical Pull Model

- Consumers are initiators of events
- Suppliers passively wait to get events pulled from them
  - They must buffer events…
- Event channel plays role of procurer



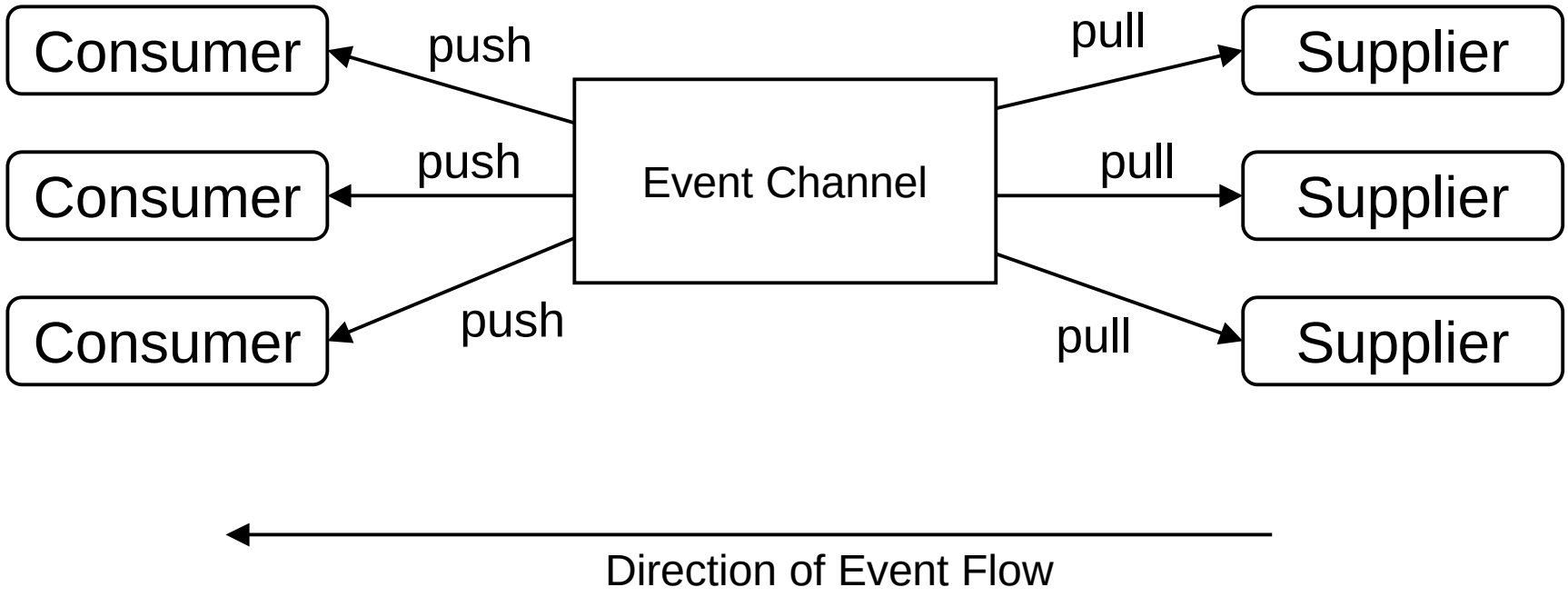
#3: Hybrid Push/Pull Model

- Suppliers push events to the event channel
- Consumers pull events from the event channel
- Both suppliers and consumers are thus active!
- Event channel plays role of a queue



#4: Hybrid Pull/Push Model

- Event channel pulls events from suppliers
- Event channel pushes events to consumers
- Both supplier and consumer are passive
- Event channel functions as an intelligent agent
  - Needs to know info about the supplier: how often events produced etc



Comparison

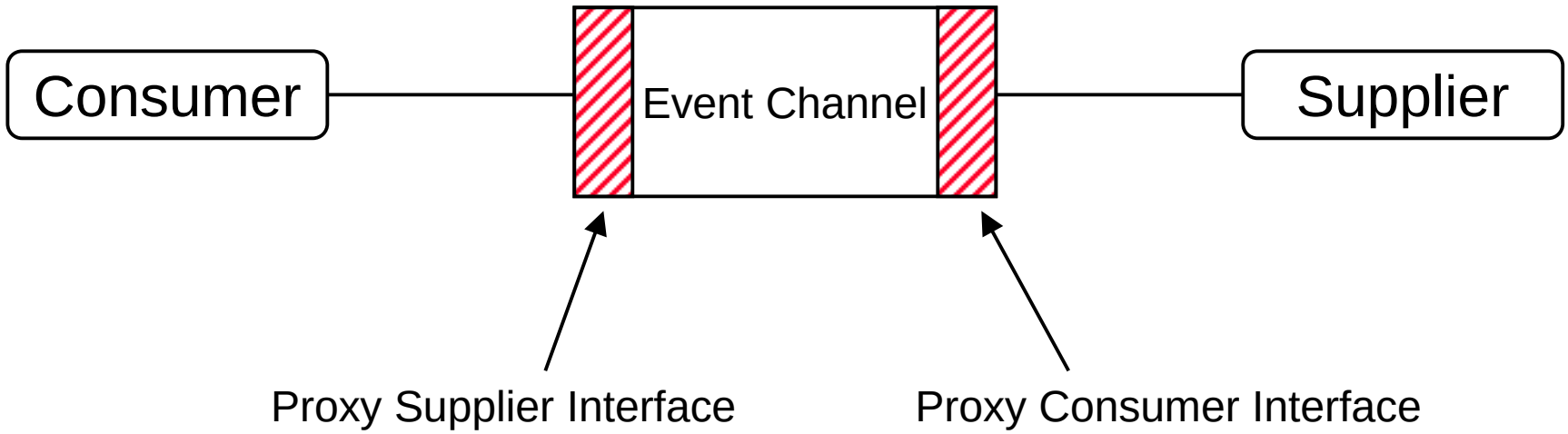
Model	Action	EC Role	Producer	Consumer
Canonical Push	Supplier pushes to EC, EC pushes to Consumer	Notifier	Active	Passive
Canonical Pull	Consumers pull from EC, EC pulls from Supplier	Procurer	Passive	Active
Hybrid push/pull	Supplier pushes to EC, Consumer pulls from EC	Queue	Active	Active
Hybrid pull/push	EC pulls from supplier, EC pushes to Consumer	Intelligent Agent	Passive	Passive

Notes on Event Service

- A single event channel can support all four models simultaneously
- Note: each consumer receives all events provided by all suppliers

Event Service Interfaces

- CosEventComm interface provides IDL to interact with event channels
  - Note: most interfaces deal with suppliers and consumers, not EC



Interfaces for the Push Model

- Push consumer implements the PushConsumer interface and registers for it with a supplier (details later…)

```
module CosEventComm {
    exception Disconnected {};

    interface pushConsumer {
        void push(in any data) raises (Disconnected);

        void disconnect_push_consumer();
    }

    interface PushSupplier {
        void disconect_push_supplier();
    }
    // …
```

Interfaces for the Pull Model

```
module CosEventComm {
    // …
    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();
    }
    // …
```

Event Channel Administrative Interfaces

```
module CosEventChannelAdmin {
    interface ProxyPushSupplier;  interface ProxyPullSupplier;
    interface ProxyPushConsumer;  interface ProxyPullConsumer;

    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();
    }
    // …
}
```

Using the Event Channel

- Consumers
  - Invoke for_consumers() on EC to obtain ConsumerAdmin object reference
  - If push consumer, invoke ConsumerAdmin->obtain_push_supplier()
  - If pull consumer, invoke ConsumerAdmin->obtain_pull_supplier()
- Suppliers
  - Invoke for_consumers() on EC to obtain SupplierAdmin object reference
  - If push supplier, invoke SupplierAdmin->obtain_push_consumer()
  - If pull supplier, invoke SupplierAdmin->obtain_pull_consumer()
- Discuss next
  - Example VisiBroker handouts: PushModel.C & PushView.C
  - Project #4 Description (handed out Wednesday 11/15)
    - It will involve the canonical push model, much like PushModel.C and PushView.C