Administrative Notes

• Project #3 will be given out Tuesday, Nov. 2
  – Will be due Tues. Nov. 30
  – Will use CORBA Event Services (discussed today)
  – If you are concerned about finishing this project, START TODAY on familiarizing with Event Services
  – Most of this lecture is on generic CORBA Event Services, but ORBACUS has some proprietary extensions we will use (factory to allow multiple instances of an event channel)

• Project #4 will overlap with Proj. #3
On Device Discovery and Mobile Computing...
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 apps**
  - Even asynchronous invocations only help some

- **Desirable flexibility**
  - **Spatial decoupling**: “clients” and “servers” do not have to know each other’s identities
  - **Temporal decoupling**: “clients” and “servers” do not have to have overlapping lifetimes

- **Another kind of Middleware**: event-based, “Message-Oriented Middleware” (MOM), “pub-sub”
  - Provides spatial decoupling, more flexibility, sometimes even temporal decoupling (if have persistent queues)
CORBA & Events

• **Supplier**: entity producing the information of interest
  – AKA “publishers”

• **Consumer**: entity receiving and using the information of interest
  – AKA “subscribers”

• **Q**: which is a “client”? which is a “server”? Why?

• Suppliers can send messages to one or more consumers with a single call

• Suppliers and Consumers are decoupled spatially: they are not aware of each other’s identity
Publish-Subscribe (Event) Architectures

- Publish
- Subscribe
- Dissemination

Message Bus
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 spatially, unaware of each other (not referring to each other)
Invocations and Events Contrasted (cont.)

• 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

• Invocations: syntactic checking and type safety
  – Invocations are type checked because method’s IDL describes all data

• Events: data needs to be self-describing
  • So types not checked while being passed
  • Consumers have to establish the type
    – Case 1: consumers know what type of data to expect
    – Case 2: consumers inspect the self-describing object to see type
  • Use IDL any type
Event Service Basics

- Orbacus man. Ch. 13 and Adv. CORBA w/ C++ Ch. 20
- 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

![Diagram of event flow](image_url)

**Direction of Event Flow**
#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**

![Diagram of Canonical Pull Model](image_url)
#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.

![Diagram of Hybrid Pull/Push Model](image-url)
# Comparison

<table>
<thead>
<tr>
<th>Model</th>
<th>Action</th>
<th>EC Role</th>
<th>Producer</th>
<th>Consumer</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Canonical Push</strong></td>
<td>Supplier pushes to EC, EC pushes to Consumer</td>
<td>Notifier</td>
<td>Active</td>
<td>Passive</td>
</tr>
<tr>
<td><strong>Canonical Pull</strong></td>
<td>Consumers pull from EC, EC pulls from Supplier</td>
<td>Procurer</td>
<td>Passive</td>
<td>Active</td>
</tr>
<tr>
<td><strong>Hybrid push/pull</strong></td>
<td>Supplier pushes to EC, Consumer pulls from EC</td>
<td>Queue</td>
<td>Active</td>
<td>Active</td>
</tr>
<tr>
<td><strong>Hybrid pull/push</strong></td>
<td>EC pulls from supplier, EC pushes to Consumer</td>
<td>Intelligent Agent</td>
<td>Passive</td>
<td>Passive</td>
</tr>
</tbody>
</table>
Notes on Event Service

• A single event channel can support all four models simultaneously
• Note: each consumer receives all events provided by all suppliers
  – OMG Notification Services has much fancier filtering
  – ORBACUS has (non-standartized, i.e. proprietary) Event Channel Factories (interface EventChannelFactory, Sec 13.4.4 in ORBACUS manual) that allow creation of different instances of an Event Service
Event Service Interfaces

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

![Diagram](image_url)

- Consumer
- Event Channel
- Supplier
- Proxy Supplier Interface
- Proxy Consumer Interface
Interfaces for the Push Model

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

```java
module CosEventComm {
    exception Disconnected {
    }

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

    interface PushSupplier {
        void disconnect_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();
   }

   // …
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_suppliers() on EC to obtain SupplierAdmin object reference
  – If push supplier, invoke SupplierAdmin->obtain_push_consumer()
  – If pull supplier, invoke SupplierAdmin->obtain_pull_consumer()
CORBA any

- A CORBA any can contain a value of any CORBA type
  - Self-describing: implementation contains value plus
    typecode describing the type of the value
  - Analogous to: Java Object type
- Purpose: send and recv. values who’s type is not
  known at compile time
- Dynamic (strong) typing: addition flexibility, additional risk
  - Won’t use a value incorrectly
  - May suffer run-time type error
Using CORBA::Any

- Using any: use operations
  - insert (<<=)
  - extract ( >>=)

- Example
  
  ```cpp
  CORBA::Any a;
  a <<= (CORBA::Long) 99; // insert 99 into a
  CORBA::Long val;
  if (!(a >>= val) abort(); // extract value from a to val
      // exact type match req'd
  a <<= "Hello, World"; // deep copy; delete prev. value
  const char *msg
  if (!(a >>= msg)) abort(); // extract; a still "owns" string
  ```
CORBA Any and C++ Ambiguity

• Problem: some CORBA types may be mapped onto same C++ type
• Solution: use the Any Helper types defined in C++
  
  ```cpp
  CORBA::Any a;
  CORBA::Boolean b = 0;
  CORBA::Char c = ‘z’;
  CORBA::Octet o = 0xff;
  a <<= CORBA::Any::from_boolean(b);  // works
  a <<= CORBA::Any::from_char(c);  // works
  a <<= CORBA::Any::from_octet(o);  // works
  a <<= CORBA::Any::from_boolean(c);  // “works”
  – no compiler complaints but will not do what you want
  ```

• Corresponding helpers for extraction
What if you don't know the type?

• Type-case pattern:

```cpp
CORBA::Long l;
if (a >>= i {
    // it was a long
    ...
} else if (a >>= CORBA::Any::to_char(c) {
    // it was a char
    ...
} else if (a >>= CORBA::Any::to_octet(o) {
    // it was an octet
    ...
}
```

This will be a key to programming your “sniffers” in Project #3
User-Defined Types and Any

• IDL compiler generates insertion and extraction operators for user-defined types: enumerations, structures, unions, and sequences

• Enumeration example:
  ```c++
  Foo::Color c = Foo::blue;
  Foo::Color cout;
  a < <= c;
  if (!(a >= cout)) (abort());
  // cout should equal Foo::blue
  ```

• No storage mgmt issues for enumerations: all by-value

• Structures, unions, sequences raise issues like strings
Any and Structures

- Given IDL:
  ```
  struct myData {
    long L1;
    string msg;
  }
  ```

- Compiler generates operations (for C++)
  ```
  void operator<(CORBA::Any &, const myData &); // by ref
  void operator<(CORBA::Any &, myData *); // by pointer
  CORBA::Boolean operator>=(const CORBA::Any &, myData * &)
  ```

- First insertion (by reference) copies value and Any now owns the location; second insertion by reference
  ```
  myData md, *mdp = ..... ; // assume mdp pointed to something real
  a <= md; //copies, containing procedure still owns md’s memory
  a <<= mdp; // copies pointer, Any owns the memory and will dealloc
              // when the Any variable a goes out of scope
              // warning: do not delete mdp here!!!
  ```

- Extraction always by pointer: Any owns, read-only so no delete
**More on Any**

- Can insert object references just like a user-defined type
- Can insert an Any into another Any
  - Similar memory management issues will arise
  - Consult Chapter 15 “Advanced CORBA Programming with C++” if want to use
- Any defined in
  - `/net/niflab/orbacus/OB-4.1.2/ob/include/OB/Any.h`
- Some operations dangerous without type checking:
  ```
  const void* value();
  ```
- Extracts C++ *(void *)* generic pointer, but then you can cast it to any other type unsafely
Example Event Service and Any Push Supplier

CosEventChannelAdmin::EventChannel_var e;

try {
    CORBA::Object_var obj =
    orb -> resolve_initial_references("EventService");
    e = CosEventChannelAdmin::EventChannel::_narrow(obj);
}

catch(const CORBA::ORB::InvalidName&){
    cerr << prog << " can't resolve 'EventService'" << endl;
    exit(1);
}

if(CORBA::is_nil(e)){
    cerr << prog
    << " 'EventService' is not an EventChannel object "
    << "reference"
    << endl;
    exit(1);
}
Example Event Svc and Any Push Supplier (cont)

// Get a SupplierAdmin object from the EventChannel.
CosEventChannelAdmin::SupplierAdmin_var supplierAdmin =
    e->for_suppliers();

// Get a ProxyPushConsumer from the SupplierAdmin.
CosEventChannelAdmin::ProxyPushConsumer_var consumer =
    supplierAdmin->obtain_push_consumer();

// Connect to the ProxyPushConsumer as a PushSupplier
// (passing a nil PushSupplier object reference to it because
// we don’t care to be notified about disconnects).
consumer->connect_push_supplier(CosEventComm::PushSupplier::_nil());

// Create an event (just a string in this case).
const CORBA::String_var eventData = CORBA::string_dup("Hello, world.");
while (1) {
    // Insert the event data into an any.
    CORBA::Any any;
    any <<= eventData;
    // Now push the event to the consumer
    consumer->push(any);
sleep...
}
Example of Event Service and Any Push-Consumer

// Derive any consumer from POA_CosEventComm::PushConsumer
class EventConsumer_impl : public virtual
   POA_CosEventComm::PushConsumer
{
    ...
};

void EventConsumer_impl::push (const CORBA::Any & data)
   throw(CORBA::SystemException)
{
   // Extract event data from the any (assume string).
   const char* eventDataStr;
   if (data > >= eventDataStr)
   {
      cout << "push received event " << eventDataStr << endl;
   }
   else
   {
      cout << "Received something other than a string" << endl;
   }
}
Example of Event Svc & Any Push-Consumer (cont.)

- In main have to instantiate an EventConsumer_impl object and hook it up to supplier:

```
EventConsumer_impl* consumerImpl = new EventConsumer_impl();
EventConsumer_var consumer = consumerImpl -> _this();
// Get a ConsumerAdmin object from the EventChannel.
CosEventChannelAdmin::ConsumerAdmin_var consumerAdmin =
    e->for_consumers();
// Get a ProxyPushSupplier from the ConsumerAdmin.
CosEventChannelAdmin::ProxyPushSupplier_var supplier =
    consumerAdmin->obtain_push_supplier();
```
Example of Event Svc & Any Push-Consumer (cont.)

// Connect to the ProxyPushSupplier, passing our PushConsumer object
// reference to it.
supplier->connect_push_consumer(consumer);

// Activate the POA via its POAManager.
PortableServer::POAManager_var poa_manager =
    poa->the_POAManager();

poa_manager->activate();
cout << "Ready to receive events..." << endl;

// Enter the ORB event loop.
orb->run();