CS3301 Practical 3

170011474

2020 May

1 Q1

1. **Component Technology** is to provide components that isolate and encapsulate specific functionalities and offer them as services in such a way that they can be adapted and reused. **Components** usually consists of multiple objects and thus have a larger granularity, which enables them to combine functionalities of the objects and offer them as a single service. Following paragraphs illustrate aspects of a component system with Android.

2. **Component Population** is a set of all functions and all interfaces needed in an application. In this case, Android has some component kinds (roles): activities, services, content providers, intents, broadcast receivers, widgets and notifications.

   (a) **Activity** normally refers to one screen

   (b) **Service** is an application component performing long-term running in the background (xx) without a user interface.

   (c) **Content providers** can access a central data repository, primarily supposed to be used by other applications.

   (d) **Intent** Intent is a messaging object to start an activity, start a service. Besides, a broadcast can be sent by passing intents. There are 2 types of intents: explicit intent is that you know who you exactly wants to wire to. The implicit way is that you do not know the target name but wire to components that have some features by intent filters. For example, if you want to call your friend then would do it by finding his explicit name. Broadcast is a message spreading out when an event occurs, which is received by apps. For example, when your device boots up then a broadcast is sent to all apps.

   (e) **Broadcast receivers** receives broadcasts. A receiver is registered in the activity. It acts like subscribers in pub-sub systems.

   (f) **Widgets** are quick views of your app's functionality and data which can be embedded into other applications. Information widgets display only important information to users, such as a widget showing time and weather condition. Collection widgets scroll in top-to-down direction. Collection of information of same type and then enabling user to open any one of them to full detail, such as an

email app which display all mails in the inbox. Control widgets display the most frequently used functionalities which user might wan to control from home screen. For example, in a audio player app, you can pause, play and stop. Hybrid widgets combine all of above three.

(g) **Notification** keeps user aware of events going on. For example, users would be notified when new messages received in facebook.

3. **Component Framework** concepts in Android:

   (a) *It helps to deploy components into their roles respectively.* In this case, the framework of android allocates different roles: activity, service, content providers and broadcast receivers, etc. to different components.

   (b) *It provides global services for usage*: for example, intents and filter mechanism for locating other components as well as accessing functions offered by other components' interfaces; providing `context` of the current global information of the environment.

   (c) *The framework provides interfaces.* In this case, each Android fragment needs to provide an interface, and customized objects, such as a customized listener, need to obey and override its original functions in its interface. Besides, each activity has a well-defined lifecycle and an interface with many functions declared. They can be overriden, such as `onCreated()`.

   (d) *The framework ensures all dependencies are satisfied.* In this case, `manifest.xml` helps to register and `build.gradle` to include external dependencies and libraries.

4. A **wiring strategy** is to choose which to be performed next. For a system to work, each component's required interfaces have to be matched with the provided interfaces of another. Wiring is the process to do this and it involves 2 stages: **deciding** components which provide required interfaces from components population available; **binding** to them and handling any unsatisfiable bindings. In android, it mainly focuses on structural type compatibility: interfaces respond to the same operations or messages.

   (a) How to wire: wiring methods contain explicit and implicit ways. In Android, at installation, each new app registers its intents with the operating system. New providers become known when they are installed. Requesting an intent causes a wiring process between client and potential providers. The detailed definition and explanation of intents can be found in 1(d).

   (b) **Dependency injection**: different components usually depend on others and exactly which components they are depending on are injected into the component system from outside. In android, this can be achieved:

      i. By **setter**: a button can call `setOnClickListener()` with a listener as a parameter to be passed into the object.

      ii. By **constructor**: when creating a new intent, the `context` can be injected as a parameter: `Intent i = new Intent(getApplicationContext(),...);`

iii. By **interface**: Android allows to create **factory** objects to obtain expected objects. The service locator design pattern in Android improves decoupling of classes from concrete dependencies. The service locator stores dependencies and then provides those dependencies on demand.

(c) When to wire:

i. in Android, it can wire **dynamically** with rewiring and re-binding when conditions change. In Android, a button can invoke one activity at first and then invoke another activity later on.

ii. It can be wired at **start-up** by updating the `build.gradle` file. Local libraries and remote libraries can be added. Re-binding only requires a restart.

2 Q2

2.1 Description of BPEL Diagram

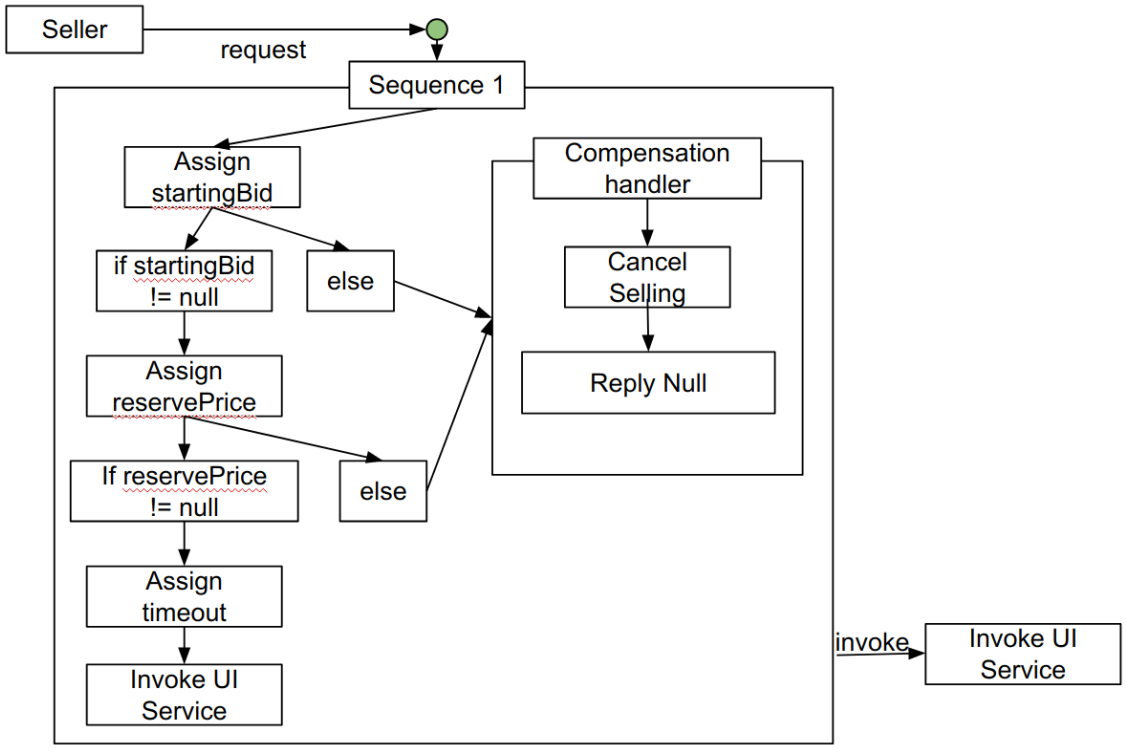

Figure 1: Selling Service: Sequence 1

Selling Service: foundation for the auction system

The Sequence 1 describes a preparation procedure, including set up `startingBid` and `reservePrice` from the request. `timeout` is set by the main process, indicating the total time to wait for receiving bids. If `startingBid` or `reservePrice` is not provided, the selling is cancelled. The `UI Service` mainly just use a UI to show there is a new item to be sold so potential buyers who are visiting the website can bid. `UI Service` is not really required since buyers can still bid without UI.

The Sequence 2 describes a receiving bidding procedure. Whenever buyer gives new bidding then the new one is pushed into a queue. The left activity is to receive bids and push bids into a queue. The timer is set. If no bids have been received for a period of time (`interval timeout`), then bidding ends. The right activity is also a timer for bidding, but this is the total amount of time for the whole bidding procedure.

The Sequence 3 is for getting highest bid. It compares each bid from the queue and get the highest bid and the buyer of the highest bid.

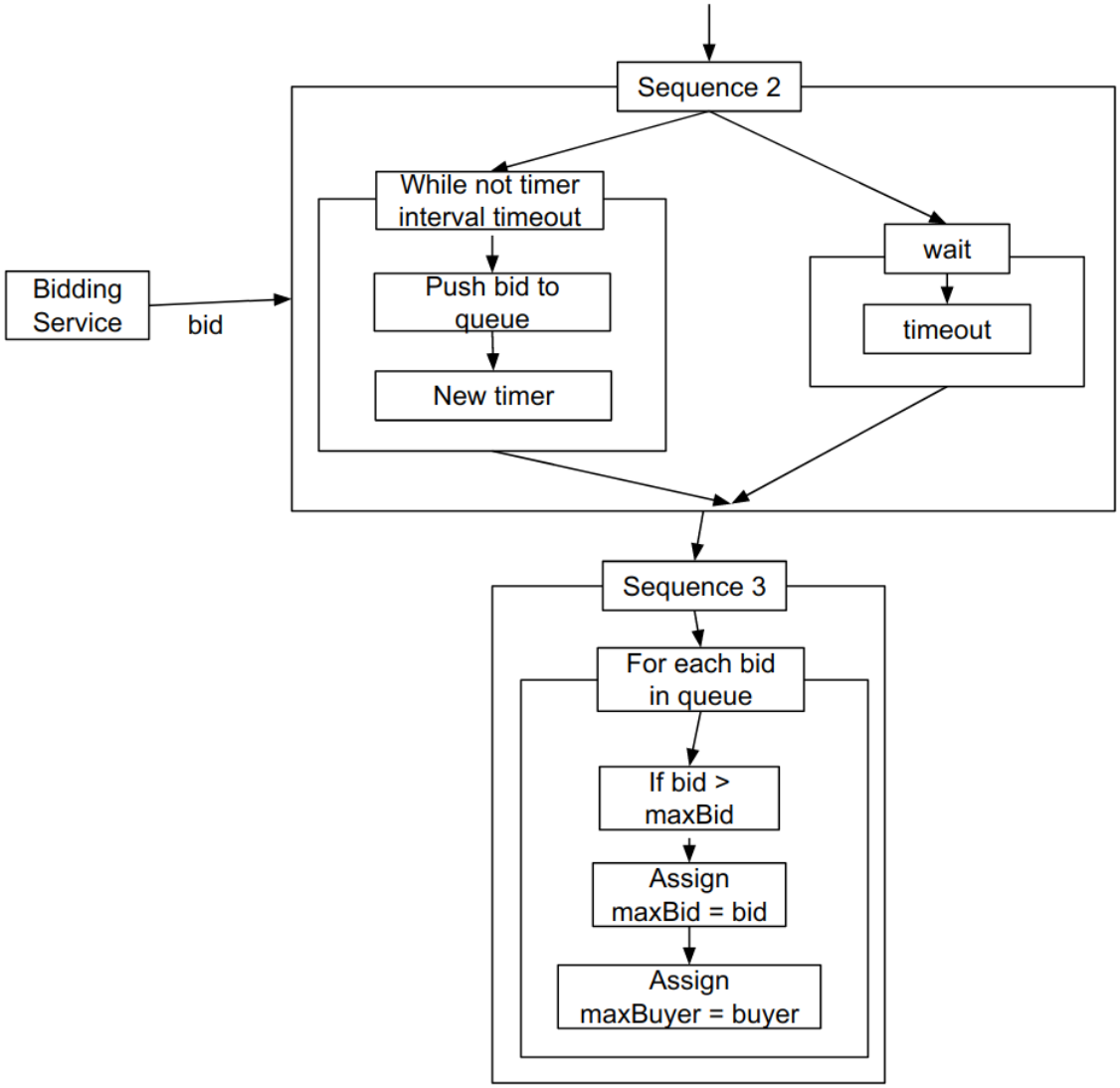

Figure 2: Selling Service: Sequence 2, 3

The Sequence 4 describes a decision-making procedure. If the highest bid does not reach the expected `reservePrice`, then `Null` is replied, indicating there is no valid buyer. Otherwise, `Bank Service` would be invoked to reserve credits. If the bank operation is successful, then the final response is returned to the seller client. Otherwise, a compensation handler is invoked to undo (remove the current `maxBid` in queue) and start to pick another highest bid from the queue.

Bidding Service

A buyer client can make a request and start the main process of Bidding Service. `bidPrice` from the request is set and send to `Selling Service`. A callback would be returned to show whether the bidding is successful or not and then a reply about the bidding result is

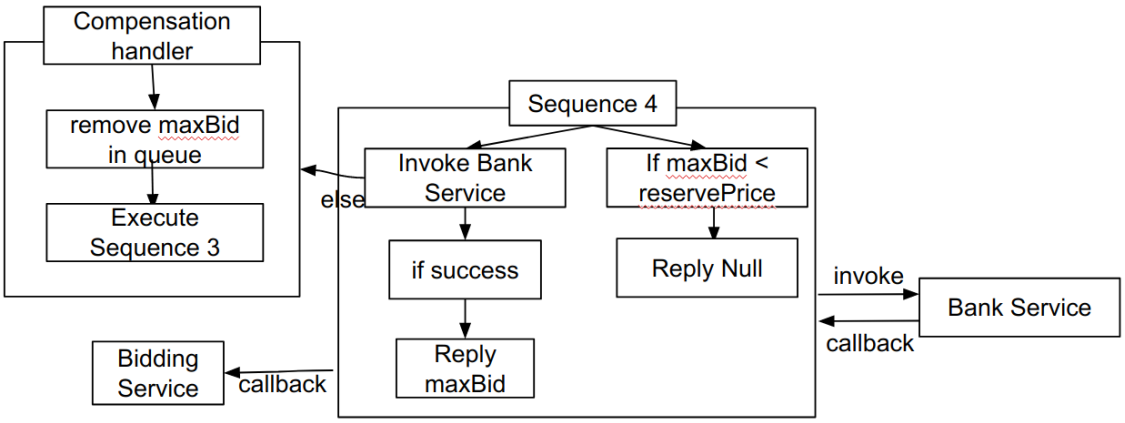

Figure 3: Selling Service: Sequence 4

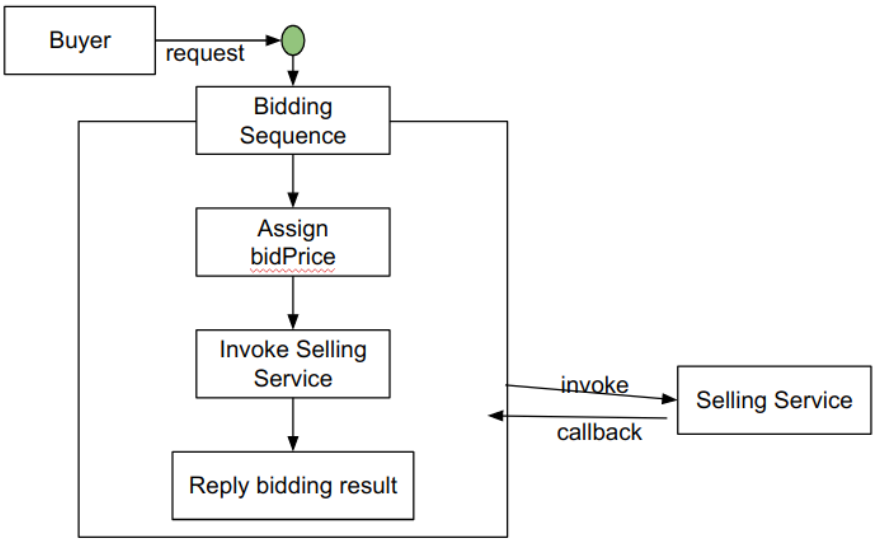

Figure 4: Bidding Service

returned to the client.

2.2 Partner

Partnerships means that a business process can have multiple partner links to provide services. Those named participant services provide well-defined services for the BPEL process. They are required by the BPEL process to function properly and they are bound at some point to actual instances (executable). Partner links are defined as communication exchanges between all the partners with which the BPEL Process interacts. It assumes that all services needed are provided by those partners. It reveals that the control does not necessarily lie in one but an integration of a supply chain. It also reflects business logic that other companies might be service providers as well.

For Selling Service, the partner is Bidding Service, UI Service, and Bank Service.

1. Bidding interface: Required by selling process, provided by Bidding Service.
2. UI interface: Optional to selling process, provided UI Service.
3. Bank interface: Required by selling process, provided by Bank Service.

2.3 Concurrency

Concurrency issues: strong synchronisation can result in failures easily and reduce possibility to schedule interactions. It also brings complexity for end-user programming since they need to know how to use mutex or semaphores to synchronise.

In this case, when buyer bids, the BPEL main process needs to receive all bids and get the highest bid among all. Bids from buyers would be pushed into a queue and BPEL process would pull a bid price from the queue and compare it with the current highest bid as the diagram shows. In this way, concurrency is managed by a `while` and `for` loop in Sequence 2. Besides, activities in Sequence 2 execute concurrently.

2.4 Dependency

Dependency would cause cascading effects so other sequences may be affected. Since the dependency also requires that effect is closely coupled with the cause, it would be less flexible. To show dependency, the BPEL process could use structured activities to impose conditional flows, such as `if`.

In this case, to decide the maxBid, it needs to compare reserve price in previous steps with the current maxBid. When the seller makes a request, if and only if all information is provided then it would be processed. So the following steps depend on the current step.

2.5 Compensation

A compensation handler helps to rollback or undo the transactions that are completed in the previous steps of the BPEL process. For example, a travel planning process can include several nested transactions to book a ticket, to reserve a hotel and a car. If the trip is cancelled, the reservation transactions must be compensated for by cancellation transactions in the appropriate order.

Error handler is for error handling, it executes when exceptions thrown by an activity. Compensation handlers execute when the BPEL process encounters a Compensate or CompensateScope activity. So the error handler is triggered by a thrown exception, and the compensation handler is triggered by a Compensate or CompensateScope activity. The error handler aims to correct the situation such that regular processing can continue outside the scope or alternate ways to complete the process can be taken. All of this might require undoing actions that have already been completed within the scope. Thus, an error handler of a scope may start the compensation handlers of its nested completed scopes to undo their actions.

In this case, when the seller does not specify `reservePrice` or `startingBid`, then the selling would be terminated and undo by a compensation handler. Besides, when Bank Service does not successfully reserve credits, then the handler would undo the current `maxBid` and rollback from Sequence 4 to Sequence 3 to find the second highest bid.

3 Q3

3.1 Terms

1. A *stub* is a mechanism that effectively creates and issues requests on behalf of a client. It represents the remote object. Calls to the stub are sent over the network to the skeleton.

2. A *skeleton* is a mechanism that delivers requests to the CORBA object implementations

3. An *object adaptor* is an object that adapts the interface of an object to the interface expected by a caller. The lack of an object adapter would mean that object implementations would connect themselves directly to the ORB to receive requests.

4. *Inter-ORB protocol* is a ORB interoperability architecture that provides for direct ORB-to-ORB interoperability. GIOP stands for general inter-ORB protocol which specifies transfer syntax and a standard set of message formats for ORB interoperation. IIOP specifies how GIOP is built over TCP/IP transports.

5. *Dynamic Invocation Interface (DII)* supports dynamic client request invocation, that is, rather than calling a stub routine that is specific to a particular operation on a particular object, a client may specify them at runtime (obtain those information from IR).

6. *Dynamic Skeleton Interface (DSI)* provides dynamic dispatch to objects. Just as the DII allows clients to invoke requests without having access to static stubs, the DSI allows servers to be written without having skeletons for the objects being invoked compiled statically into the program.

7. *Interface repository (IR)* allows the IDL type system to be accessed and written programmatically at runtime. The IR is itself an object whose operations can be invoked just like any other CORBA object. By using it, applications can traverse an entire hierarchy of IDL information. The purpose of the interface repository is to maintain type information about IDL files. Once an IDL file is compiled, its definitions can be stored in an interface repository and can be retrieved remotely by other ORBs.

3.2 Static Approach (Stub and Skeleton)

To create it:

1. Programmers create/acquire IDL file describing the interface.

2. Create object implementation and instantiate.

3. Save the resulting IOR.

4. Tell the object adapter about the new object which creates a skeleton.

5. Client uses IDL file to generate interfaces.

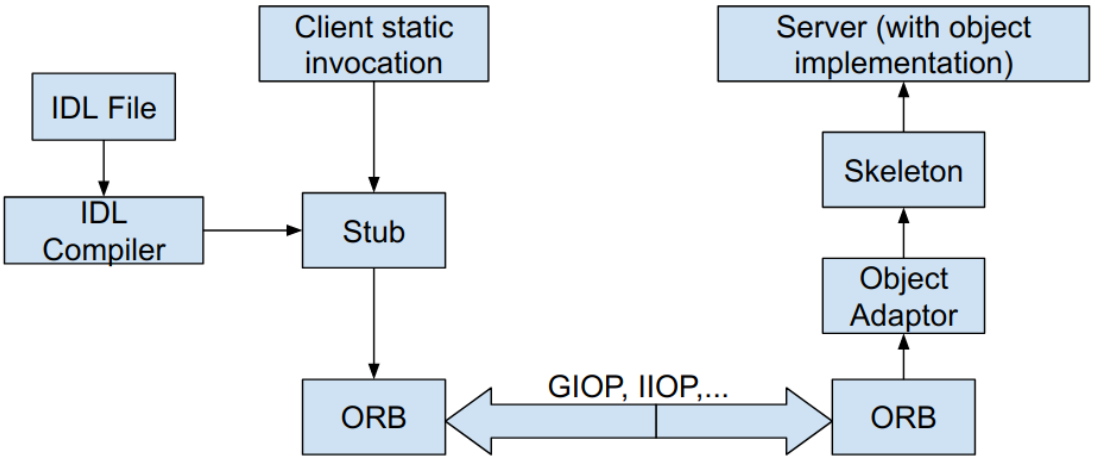

Figure 5: Static Approach

6. Clients create a new stub using interface and IOR.

To make a call:

1. Clients make calls to its stub. The stub directly works with the client ORB to marshal the request. That is, the stub helps to convert the request from its representation in the programming language to one suitable for transmission over the connection to the target object via an inter-ORB protocol.

2. Once the request arrives, the server ORB and the skeleton cooperate to unmarshal the request.

3. The object adaptor adapts the interface to the interface expected by the caller. For example, one object can have multiple object adaptors due to different programming language, object lifetimes, policies and so on. A single object adaptor is difficult to serve all.

4. The request is dispatched to the object through skeleton and once the object completes the request, any response is sent back the way it came.

3.3 Dynamic Approach (DII and DSI)

To make a dynamic call:

1. A client application (usually a foreign object) can invoke requests on any object without having a static stub. It creates a request object through `create_request` operation provided by CORBA::Object interface. Since every IDL interface is derived from CORBA::Object, every object automatically supports `create_request`. By calling it on an object reference for the target object, an application can create a dynamic request.

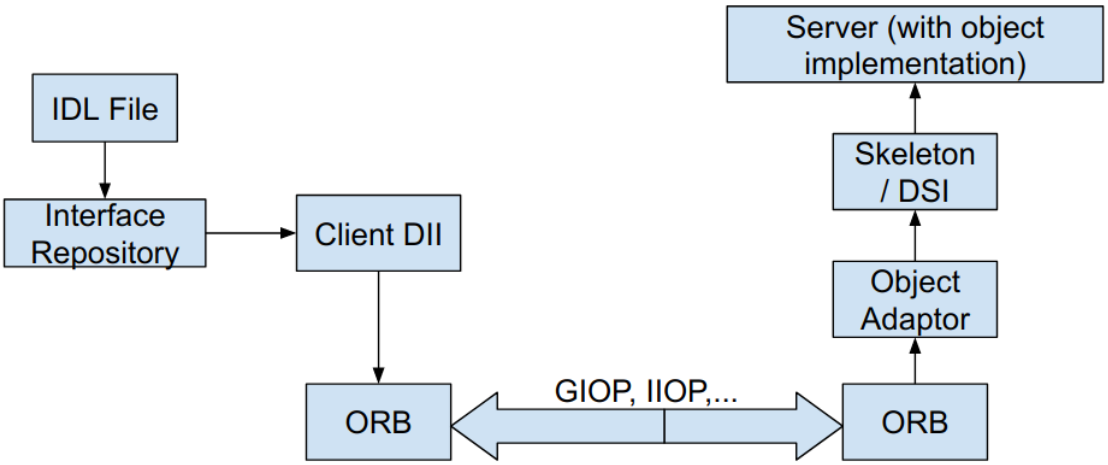

Figure 6: Dynamic Approach

2. Arguments are provided and the types of them are determined by IR.
3. Invoke the request in 3 possible ways:
   (a) Synchronous invocation: the client invokes the request and then blocks waiting for the response.
   (b) Deferred synchronous invocation (only dynamic invocation has): the client invokes the request, continues processing while the request is dispatched, and later collects the response.
   (c) One-way invocation: the client invokes the request, and then continues processing; there is no response.
4. The rest of the steps on the server side are the same as ones of the static approach.

The DSI is analogous to DII in the server side. Just as the implementation of an object cannot distinguish whether its client is using type-specific stubs or the DII, it makes no difference for the object and it is not even perceptible whether an invocation was triggered through a compiler-generated skeleton class or through the DSI. It receives requests without requiring that static skeletons for each object's interface type be compiled into it.

3.4 Compare and Contrast

1. Both perform a request by gaining access to a reference for a CORBA object and invoking the operation that satisfies the request. The CORBA server cannot tell the difference between them.
2. Static approach is easier because the CORBA client invokes operations on the client stubs.
3. Only dynamic approach uses IR at the runtime.

4. Only dynamic approach has deferred synchronous invocation, although later on the OMG issued an RFP which provided it for ORB.

5. Static approach is for compiling time (the target is explicitly known) and dynamic approach is for request in running time (the target can be unknown in compiling time).

6. Dynamic approach brings more flexibility, since free of skeletons and stubs brings convenience for foreign objects.

7. Dynamic approach brings more complexity since programmers need to take jobs of the static stub and static skeleton.

8. The creation and invocation of a single DII request could actually require several actual remote invocations, making a DII request several times more costly than an equivalent static invocation. The reason is that creating a DII request may cause the ORB to transparently access the IR to obtain information about the types of arguments and return value. Since the IR itself is a CORBA object, each IR request is indeed a remote invocation as well.

4 Q4

4.1 Definition of Linguistic Reflection

Linguistic reflection is the ability of a running program to generate new program fragments and to integrate these into its own execution, allowing a program's behaviour to adjust dynamically in order to provide flexibility and high productivity.

4.2 Examples

The traditional way to build stubs and skeletons is by IDL compiler in a static manner. However, the dynamic generation can be achieved by reflection.

For example, the following figure contains XML-RPC messages for a method `circleArea` with a double type parameter `radius`, and a method `circleCircumference` as well. The `circleArea` is to get a circle's area given a radius, and `circleCircumference` is to get a circle's circumference given a radius:

```
<!-- The HTTP headers for these requests will reflect the senders and the
    content. -->
POST /xmlrpc HTTP 1.0
User-Agent: myXMLRPCClient/1.0
Host: 192.168.124.2
Content-Type: text/xml
Content-Length: 169
<?xml version="1.0" ?>
<methodCall>
    <methodName>circleArea</methodName>
    <!-- parameters to be passed -->
    <params>
        <param>
            <value>
                <double>2.41</double>
            </value>
        </param>
    </params>
</methodCall>

POST /xmlrpc HTTP 1.0
User-Agent: myXMLRPCClient/1.0
Host: 192.168.124.2
Content-Type: text/xml
Content-Length: 169
<methodCall>
    <methodName>circleCircumference</methodName>
    <!-- parameters to be passed -->
    <params>
```

```
        <param>
            <value>
                <double>2.41</double>
            </value>
        </param>
    </params>
</methodCall>
```

Without Linguistic Reflection

In static stubs and skeletons, XML-RPC message includes method name as well as parameter values with their types, and the server side needs to know all methods statically to parse the message.

```
public class Decoder{
    // statically
    void makeXMLRPCCallWithCompiler(Request req){
        if(req.getMethodName().equals("circleArea")){
            double v = getIntegerValue(req);
            // a is the object which can really do things
            a.circleArea(v);
        }
        else if(req.getMethodName().equals("circleCircumference")){
            double v = getDoubleValue(req);
            a.circleCircumference(v);
        }
    }
}
```

with Linguistic Reflection

In a dynamic way, XML-RPC message does not have to know methods it has and match it by the explicit name like `equals(\circleArea")`. Instead, it can directly extract the method name and parameter classes(types here) from the request. Afterwards, those extracted information are used as parameters to construct a method Class object and then it is invoked by the real parameter values. In this way, there is no need to write many cases switching between different method name and remember parameter types respectively:

```
public class Decoder{
    // dynamically
    void makeXMLRPCCallWithReflection(Request req){
        // an object representing the class of a
        Class aClass = a.getClass();
        // an object representing the method on a
        Method m = aclass.getMethod(req.getMethod(), req.getParameterClasses());
        m.invoke(a, req.getParamaterValues());
```

```
    }
}
```

4.3 Evaluation of Linguistic Reflection

Benefits

1. Load new components at runtime with re-compiling: in XML-RPC, if a server is already compiled and afterwards a new method is integrated, there is no need to compile again with huge time waste for generating a new case (i.e. `equals(\newMethod")`) in a if else statement, since the reflection would automatically load the new method name and parameters to find the matched method.

2. Extend or replace functionality of a running system: in XML-RPC, same as above, an existing method can be overwritten or updated.

3. No loss of type-safety: in XML-RPC, you still can't call a method on an object that doesn't implement it: in XML-RPC, if the expected object implementation does not contain the matched method signature in the request, then no call is made.

4. The skeleton can be more generic and is easier to be made and the server can be more flexible. In XML-RPC, since the call can be made even if the method is not available at compiling time. Less concrete knowledge is required for all methods.

Drawbacks

1. Late error detection: the error would not be shown in compiling time until at the runtime you find that you fail to get a method. In XML-RPC, even if there is an error inside, it only occurs when you have made the request from the stub.

2. Security and broken encapsulation: other components can get the metadata of yours which is not hidden anymore. In this case, you can know all private and public methods.

3. On the client side, using reflection to build stubs is more complicated, because directly using a middleware or use an IDL compiler to make calls is easier, and also it needs to create each method using reflection for the stub compiler as well. If you abuse it, you can turn an otherwise simple problem into a complex and ugly mess.

4. Java does not provide complete access to its meta-level so no calls can be intercepted.