A GENERIC DATA DISPATCHER FOR CORBA-BASED APPLICATIONS

Markus Aleksy, Ralf Gitzel, Martin Schader
Department of Information Systems
University of Mannheim
Schloß (L 5,6)
68131 Mannheim, Germany
{aleksy|gitzel|mscha}@wifo3.uni-mannheim.de

Abstract

In this paper we present a generic data dispatcher which can distribute data without any requirements with regard to the data format. Two major topics to be considered are determination of the actual message type and dynamic creation of sub-sequences which can be sent to different servers. An example based on the CORBA framework was implemented to prove the concept.

Key Words

Data dispatching, dynamic type handling, CORBA

1. Motivation

The development of distributed applications suffers from two major restrictions – network bandwith and computational capacity. The latter can be addressed by a *dispatcher* which assigns the data packages to different *workers* which in turn solve the sub-problems assigned to them in parallel (Master-Slave-Pattern [1], Object Group Services/Join-Services Pattern [2], [3]). Within the CORBA context, these workers are realised by *servers*.

A dispatcher can also be used to remedy bandwidth shortage by allowing applications to make use of the optimum package size. Small messages are joined into a single one and are then separated to be distributed to their respective recipients.

A dispatcher of a generic nature can thus be used by a great number of applications.

Our example implementation is based on the Common Object Request Broker Architecture (CORBA) [4]. This standard's popularity has steadily increased ever since the release of specification 2.0, not only due to its platform and language independence, but primarily because of its interoperable system architecture. With its help, data exchange between implementations based on different vendors' CORBA products is made possible.

Of further interest is the Interface Definition Language (IDL) which is used in this paper to explain the concepts employed in an easy and programming language independent way.

2. Design goals

The following topics where our primary concern during the development of the dispatcher:

- Generality: The service was designed to allow a wide spectrum of applications to utilise it. The consequence is that the dispatcher has to support different data types and dispatching strategies.
- Strong Typing: Despite the generic nature of data transmission, the programmer is still able to discern data types to avoid runtime errors.
- Portability: the solution chosen is strictly conforming to the standard and thus not tied to a specific ORB product or platform.

Before delving deeper into the functionality of the dispatcher, possible designs for message formats shall be introduced.

3. Data formats

To ensure the sensible distribution of data, it must be provided in an array-like structure. The structure of an array (i.e., a collection of elements) provides a natural criterion for the division into pieces to be sent to single servers. This is not necessarily the case with other data structures or even primitive types. Different strategies can be used to decide which servers will handle which elements.

CORBA provides two array-like data structures: plain *arrays* and *sequences*. Section 3.1 will address the peculiarities of the latter. Afterwards, the use of different data formats in a single CORBA method call will be covered. There are two aspects to be consid-

ered. First, a method can be used for sequences of different types. This addresses the basic dispatcher case of sending a single type of data to different machines.

What is more, sequences can also be comprised of different data types. This allows the dispatcher to provide servers with different needs with data.

3.1. Transmission of arbitrary arrays and sequences with a single type of elements

The CORBA standard encompasses the definition of various primitive data types e.g. `long` or `double`, composite data types such as `enum` and `struct`, the template types `sequence` and `string`, object types and value types.

The transfer of several elements of the same type is handled by sequences or arrays. An array is defined using a `typedef`, specifying a type and a name, and providing the number of elements in brackets. Thus an array can contain but one type.

Sequences are vectors of variable length which can contain data of an arbitrary type. A sequence can either be bounded, i.e., restricted with regard to maximum size or unbounded. The latter are only limited by memory vacancy.

Due to their greater flexibility, we will focus on sequences in the following sections. Of course, the same concepts also apply to arrays.

In order to allow different types of sequences or arrays to be transmitted, the CORBA IDL type `any` is used which is mapped to a wrapper class, `Any`, in Java. An `any` variable is able to store any type of data without loss of type information. The following code sample illustrates this feature. As one can see, arrays of `ulong`, sequences of `long`, and even matrices (sequences of sequences) can all be handled by an `any`.

```
interface Server {
  typedef unsigned long uLong[1000];
  typedef sequence<long> longSeq;
  typedef sequence<sequence<float> >
    floatMatrix;
  void dispatch(in any message);
};
```

Figure 1 illustrates this data format. The elements (grey) are encapsulated in the sequence which in turn is wrapped by the `any`.

However, this approach assumes that all servers to be contacted will be able to handle the data type transmitted. Servers which expect different data types can be handled according to the technique discussed in the next section.

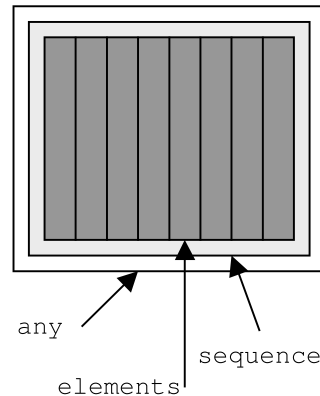

Fig. 1: Data format for the transmission of elements of the same type

3.2. Transmission of arbitrary arrays and sequences with elements of arbitrary type

In order to transfer subsequences of different types in a single sequence special measures have to be taken. In the simplest case it is sufficient to use a sequence of `any` elements as illustrated in the following IDL interface:

```
interface Server {
  typedef  sequence<any> anySeq;
  void dispatch(in any message);
};
```

The `message` argument can contain an arbitrary type. Therefore, the above-defined `anySeq`, a sequence which can hold elements of different types, can be used. Passing a message of type `anySeq` allows the inclusion of vastly different types of data according to the needs of a specific server.

Should additional information on the elements be required the above scheme can be modified to contain `structs`. The changes can be seen in Fig. 2. In addition to the data to be sent, the elements also contain information on the dispatching strategy, server references etc. The following IDL interface gives an example for this advanced version. Notable changes to the original version are the definition of the `Message struct` consisting of data and dispatch information and the corresponding sequence `mes-`

sageSeq. The dispatch method uses the latter as a parameter.

```
interface Server {
  struct Message {
    Object receiver;
    string method;
    any data;
  };
  typedef sequence<Message>
    messageSeq;
  void dispatch(in messageSeq msg);
};
```

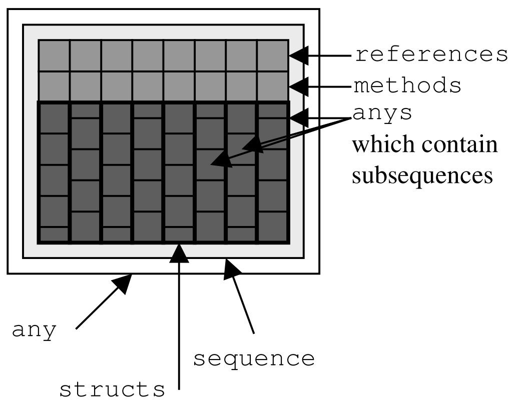

Fig 2: Data format for the transmission of arbitrary data plus additional information

This way, different types of data can be dispatched, be it primitives, arrays, sequences, or composite types.

4. Data distribution technique

Realising the dispatcher described above in a CORBA environment requires the solution to two major obstacles inherent to the problem. These are:

- sequence recognition and
- creation of sub-sequences based on a distribution strategy.

4.1. Sequence recognition

Despite CORBA's strong typing, it is impossible for the dispatcher to readily recognise all types it is required to handle because it is possible to define aliases via typedefs. Consider the following example:

```
module typetest {
  typedef sequence<long> longSeq;
  struct stru {
    double d;
    unsigned long ul;
  };
  typedef sequence<stru> structSeq;
  interface iface {
    void op();
  };
  typedef sequence<iface> ifaceSeq;
  interface Dispatcher {
    void dispatch(in any message);
  };
};
```

Without further analysis, it is impossible to discern an alias such as longSeq, structSeq, or ifaceSeq as being a sequence, e.g., recognising longSeq as a sequence of longs. Yet, since the dispatcher is only able to handle the latter, it must somehow interpret the alias to get to the actual type. The following Java code snippet illustrates how that problem can be solved:

```
TypeCode tc = message.type();
while(tc.kind() == TCKind.tk_alias) {
  tc = tc.content_type();
}
```

The basic idea behind this while loop is that we look at the type of the message and if it is an alias we re-establish the original type. The loop addresses the fact that there might be aliases of aliases.

In the end, a sequence should be recognised (TCKind.tk_sequence) or else an appropriate error handling has to be started.

Now that the nature of the message has been assured, sub-sequences can be generated.

4.2. Creation of sub-sequences

The creation of sub-sequences might seem trivial at first, yet there are some technical aspects to take care of. The sub-sequences are generated to be sent to the various applications that should handle the original messages. For efficiency reasons, messages to the same destination are grouped, therefore new sequences arise instead of single messages.

Sub-sequence generation consists of the following steps:

- Preparation: the data has to be extracted from the sequence to be analysed.

- Distribution: the data has to be distributed into sub-sequences according to a dispatcher strategy (see section 4.3).
- Sub-sequence generation: the new sequences have to be generated.

The major problem is that we have a sequence of elements of unknown types. Since the dispatcher is generic in nature, few assumptions can be made about these.

Both preparation and generation make heavy use of CORBA's ability to dynamically use types which have not been defined locally. The basic CORBA interfaces to use in this case are `DynAny`, `DynSequence` and `DynAnyFactory`. `DynAny` enables the construction of an `Any` at runtime, without having static knowledge of its type. Recall that `Any` is a wrapper class and therefore we need a way to access its content. Normally, it would be accessed via type-specific `Helper` classes. However, since these are statically generated by the IDL compiler they are not necessarily available to the dispatcher. `Any` values can be dynamically interpreted and constructed through `DynAny` objects. A `DynAny` object is associated with a data value, which corresponds to a copy of the value inserted into an `Any`. The `DynSequence` interface is derived from the `DynAny` interface. `DynSequence` objects are associated with sequences.

The `DynAnyFactory`, introduced in specification 2.4 [4], provides us with functionality which was originally found in the `ORB` class in CORBA version 2.3 [5]:

- `create_dyn_any`: This method creates a new `DynAny` object from an `Any` value. A copy of the `TypeCode` associated with the `Any` value is assigned to the resulting `DynAny` object. The value associated with the `DynAny` object is a copy of the value in the original `Any`.
- `create_dyn_any_from_type_code`: Dynamic creation if an `Any` involves creating a `DynAny` object using this method, passing the `TypeCode` associated with the value to be created.

Depending on the `TypeCode`, the created object may be of type `DynAny`, or one of its subtypes, such as `DynSequence`. Thus, the returned reference has to be narrowed to one of the complex types, such as `DynSequence` or `DynArray`, where appropriate.

A reference to the factory object is obtained by the following piece of code:

```
ORB orb = ORB.init();
org.omg.CORBA.Object obj =
  orb.resolve_initial_references(
    "DynAnyFactory");
DynAnyFactory da_factory =
  DynAnyFactoryHelper.narrow(obj);
```

A `DynSequence` offers, amongst others, the following methods:

- `get_elements`: This method returns the elements of the sequence as a sequence of `Any`s.
- `set_length`: This method sets the length of the sequence.
- `set_elements`: This method sets the elements of a sequence. The length of the `DynSequence` is set to the number of elements in the array passed to the method.
- `to_any`: This method is inherited from `DynAny`. It creates an `Any` value from a `DynAny` object. A copy of the `TypeCode` associated with the `DynAny` object is assigned to the resulting `Any`. The value associated with the `DynAny` object is copied into the `Any`.

Using these classes, the preparation step can be handled in the following way:

```
DynAny da_in =
 da_factory.create_dyn_any(message);
DynSequence ds_in =
 DynSequenceHelper.narrow(da_in);
Any[] as_in = ds_in.get_elements();
Any[] as_out =
 new Any[number_of_data];
```

`as_in` contains all elements from the sequence received, which have been extracted from the dynamical sequence. `as_out` is used to temporarily store the sub-sequence to be send next. The strategy by which to decide on sub-division is explained in detail in section 4.3.

After deciding which elements to include in a sub-sequence, the actual data structure (in the form of a `DynSequence` of the proper type) has to be constructed using the data from the `as_out` array. The code for this step has the following appearance:

```
DynAny da_out = da_factory.
 create_dyn_any_from_type_code(tc);
DynSequence ds_out =
 DynSequenceHelper.narrow(da_out);
```

In our case, the tc-Argument is the type code TCKind.tk_sequence, since we want a DynAny of exactly this type. After narrowing the object we end up with the DynSequence that only has to be set to the proper size (via set_length) and filled with the data from as_out using set_elements.

Note that while we now have the data structure we want to send to the applications, we still have to change the type to make it compatible with our method which requires an in parameter of type any. The above steps result in the following code:

```
ds_out.set_length(no_of_data);
ds_out.set_elements(as_out);
message = ds_out.to_any();
```

The code fragments are put into context in the example implementation in appendix A.

4.3. Dispatcher strategies

As mentioned in section 4.2, the division into subsequences is based on certain rules, which make up the dispatcher strategy. The following, which are used in our Object Group Service/Join Service (OGS/JS) project [2],[3], are to be considered core strategies, yet other rule sets may be implemented as well:

- ANYTHING: The dispatcher sends the complete elements to all workers.
- PER_ELEMENT: The elements are distributed to the servers in a round-robin fashion, one element at a time.
- SAME_SIZE: With this strategy, the dispatcher uses the value of data_length as the specification of the number of data elements to be sent to each worker. The amount of data specified in data_length is assigned to each worker, e.g., if data_length=3, the first worker gets the first three elements, the second worker gets the next three elements, and so on, until no elements are left. Should the number of elements to be transmitted exceed the product of the number of workers and data_length, the surplus elements are lost. In the opposite case, some workers might receive no data at all.
- DIFFERENT_SIZE: If the DIFFERENT_SIZE strategy is used, the data_size component will be evaluated to determine the individual quantity of elements to be assigned to each specific worker. The sequence data_size consists of an array of integers whose values reflect the number of elements the corresponding worker has to process. Thus, data_size[i] contains the number of elements to be delivered to worker i+1. It is permissible to have the value 0 contained in the sequence, in which case the corresponding worker does not get any data. Should the actual number of elements to be distributed be bigger than the sum of the integer values contained in data_length, the additional elements are simply neglected. In the opposite case, the specified quantities are being delivered as long as there are elements left. If the dispatcher runs out of data, dispatching is terminated, so that some workers might possibly remain unconsidered.
- Default: If the dispatcher is not able to recognise the data format provided (i.e., if it is not a CORBA sequence), it will use its default dispatching strategy which is equivalent to the ANYTHING strategy.

The following code snippet shows the IDL definition of the appropriate part of the message format used in our OGS/JS project:

```
enum DataDispatchStrategy {
  ANYTHING,
  PER_ELEMENT,
  SAME_SIZE,
  DIFFERENT_SIZE
};
typedef sequence<unsigned long>
  sizeSeq;
struct DataStructure {
  DataDispatchStrategy dds;
  sizeSeq data_size;
  unsigned long data_length;
  any data;
};
```

5. Conclusion

The solution presented here offers several interesting advantages. It allows parallel data processing, regardless of programming language, operating system, hardware and CORBA product utilised. Also is does not matter whether the data format was defined at development time or not.

However, it was impossible to achieve total portability. The reason for this is inherent to the CORBA standard itself. As mentioned before, the functionality of the dynamic classes such as DynAny was found in the ORB class – in the case of Java org.omg.CORBA.ORB – in version 2.3.

This, however, has changed in version 2.4 where said functionality has found its way into its own classes in the package `org.omg.DynamicAny`. Porting our approach – or any other application – from version 2.3 to 2.4 requires the mentioned changes, even if the program is completely compliant with the standard.

References

[1] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *A System of Patterns – Pattern-Oriented Software Architecture* (John Wiley & Sons, Chichester, 1996)

[2] M. Aleksy, A. Korthaus, A CORBA-Based Object Group Service and a Join Service Providing a Transparent Solution for Parallel Programming, *Proc. of the International Symposium Software Engineering for Parallel and Distributed Systems*, Limerick, Ireland, 2000, 123-134

[3] M. Aleksy, M. Schader, A Scalable Architecture for Parallel CORBA-based Applications, *Proc. of the 13th IASTED International Conference on Parallel and Distributed Computing and Systems*, Anaheim, USA, pp. 200-205

[4] OMG, CORBA/IIOP 2.4 Specification, OMG Technical Document Number 00-10-01, 2000, ftp://ftp.omg.org/pub/docs/formal/00-10-01.pdf

[5] OMG, CORBA/IIOP 2.3 Specification, OMG Technical Document Number 99-10-07, 1999, ftp://ftp.omg.org/pub/docs/formal/99-10-07.pdf

Biographies

Markus Aleksy graduated in Management Information Systems in 1998 at the University of Mannheim, Germany. Currently he is a Ph.D. student in the Department of Information Systems at the University of Mannheim. His research interests include design, implementation, and evaluation of distributed systems, especially CORBA.

Ralf Gitzel, is a Ph.D. student at the University of Mannheim, Germany, where he graduated in Management Information Systems in 2000. His research interests include distributed systems, code generation, and information visualisation.

Prof. Dr. *Martin Schader* studied Business Administration, Operations Research, and Computer Science at the University of Karlsruhe. He holds a doctorate degree from the University of Karlsruhe and obtained his habilitation at University of Augsburg. Currently he is a professor of Management Information Systems at the University of Mannheim.

Appendix A: Example Implementation

```
synchronized void distribute(Any message) {
  try {
    // get TypeCode
    TypeCode tc = message.type();

    // get real type
    while(tc.kind() == TCKind.tk_alias) {
      tc = tc.content_type();
    }

    // is it a sequence?
    if(tc.kind() == TCKind.tk_sequence) {
      // resolve data into a DynSequence
      org.omg.DynamicAny.DynAny da_in = da_factory.create_dyn_any(message);

      // cast to DynSequence
      org.omg.DynamicAny.DynSequence ds_in =
        org.omg.DynamicAny.DynSequenceHelper.narrow(da_in);

      // create Any-array to access the elements
      Any[] as_in = ds_in.get_elements();

      // create Any-array to save the new generated subsequences
      Any[] as_out = new Any[number_of_data];

      // copy the elements to subsequence
      // e.g.: as_out[i] = as_in[j];

      // create a new DynAny
      org.omg.DynamicAny.DynAny da_out =
        da_factory.create_dyn_any_from_type_code(tc);

      // cast to DynSequence
      org.omg.DynamicAny.DynSequence ds_out =
        org.omg.DynamicAny.DynSequenceHelper.narrow(da_out);

      // set number of elements in the subsequence
      ds_out.set_length(no_of_data);

      // set elements
      ds_out.set_elements(as_out);
      message = ds_out.to_any();

      // distribution of the subsequence
      // e.g.: send(message);
    }
  }
  catch(org.omg.DynamicAny.DynAnyPackage.InvalidValue ex) {
    //...
  }
  catch(org.omg.DynamicAny.DynAnyPackage.TypeMismatch ex) {
    //...
  }
  catch(org.omg.DynamicAny.DynAnyFactoryPackage.InconsistentTypeCode ex) {
    //...
  }
}
```