The Stock Quoter Case Study

Mar 27, 2009

Martin Steffen and Tran Thi Mai Thuong

Department of Computer Science, University of Oslo, Norway

Abstract. This document describes on an abstract level the *stock quoter case study*. The case study offers simple functionality for stock trading over the internet and has been used for illustrating the Corba component model [2,1], for instance in [4]. The document is intended to give an overview over the functionality of the system.

1 Introduction

The *stock quoter system* is a web-based software that supports clients interested in the stock market (henceforth called *brokers*). The software allows the brokers to keep up-to date with information about the stock market, i.e., the current prices of individual stocks or other financial products.

The stock quoter system has been used as some typical piece of component software, in particular exhibiting two different kinds of communication, roughly *event-based* communication in a publish-and-subscibe mode, and a request-and-respond comunication mechanism, based on method calls and returns. [4], for instance, uses the stock quoter system to illustrate the Corba component model and the use of IDL3.x

2 Architecture and Specification

This section describes the *architecture* of the system, i.e., its decomposition in different (types of) components, and their respective interfaces. First we give a short overview of the two main components in Section 2.1. Their interaction is described in more detail afterwards in Section 2.2, distinguishing between the request and response interaction and the event-based interaction. The interface specifications as well as the data structures are expressed in the IDL 2.x specification language from the OMG group. We start with the interfaces, as they give a good overview of the functionality, even if the types of the communicated data is not immediately provided. The remaining data types are given afterwards in Section 2.3.

2.1 Description of the components

The system, as discussed here, includes two (kinds of) components: the *stock distributor* (or distributor for short) and the *stock broker* (or broker). We assume

that there is one distributor and an arbitrary number of brokers in the system. The brokers do not communicate with each other and so the only communication is between the distributor and individual brokers.

2.1.1 Distributor The stock distributor plays the role of an information server for the brokers. It monitors a real-time data-base (not modelled here) containing the stock data and it publishes a real-time feed of stock information to the associated brokers. As said, we consider one instance of the distributor component. In real-life situations, in contrast, it is of course possible, that a broker receives information from various distributors, for instance, each one associated with a different stock exchange. The distributor offers to the brokers the possibility to *subscribe* and *unsubscribe*. Conceptionally, the distributor is administered by a controller component, which is not modelled here, and which has administrative functionality. It is used to start and stop the distributor server, and change certain parameters (for example, the notification rate).

Conceptually, the distributor offers two types of interfaces, one for the administrative controller, the other, the main one, for its clients, the brokers. The provided interface for the controller, the trigger interface of Listing 1.1, is rather simple: the controller can start or stop the distributor from performing its service. It is a non-descript interface for administering servers, and is inherited by the stock distributor interface.

Listing 1.1. Trigger

```
interface Trigger // distributor interface for the controller
{
  void start ();
  void stop ();
}
```

Listing 1.2 shows the (main part of the) distributor's interface. It inherits the trigger interface (offered to the controller). The first method represents the request/response communication mechanism, the latter two belong to the event-based communication, dealing with subscription and unsubscription.

Listing 1.2. Distributor

```
interface StockDistributor : Trigger
{
  /// request/response interaction: provided interface
  StockQuoter provide_quoter_info ();


  /// methods for event-based communication

  Cookie               subscribe_notifier (in StockNameConsumer c);
  StockNameConsumer  unsubscribe_notifier (in Cookie ck)
    raises (Invalid_Subscription);;

  attribute long notification_rate;   /// rate of updates

};
```

2.1.2 Broker The component acts a client resp. as a subscriber to the distributor. Again, there are two modes of obtaining information: a broker can actively request information from the distributor server and to receive information this way, a broker needs *not* to be subscribed. The second mode are *notification events* for subscribed brokers, informing about changes of individual stocks. Listing 1.3 shows the stock broker interface in IDL2.x, where the first three method correspond to the request/response mechanism, and the remaining one represents the event-based communication.[1]

Listing 1.3. Broker

```
interface StockBroker{


  void        connect_quoter_info   (in StockQuoter c);   // set

  StockQuoter disconnect_quoter_info ();                  // reset
  StockQuoter get_connection_quoter_info ();              // get

  /// Event-based communication (event sink)
  StockNameConsumer get_consumer_notifier_in ();


};
```

2.2 Communication mechanisms

Next we describe the interaction between a distibutor and a broker in more detail.

2.2.1 Request and response Request and response is the known interaction model of object-oriented languages or client-server type of interaction, i.e., invoking a method or service and getting a response back. This kind of interaction is used by the broker to obtain information about stocks ("quoter information") from the distributor. In component terminology, the distributor *provides* that interface and the broker on opposite side, acting as a client, *uses* or *requires* that interface.[2]

So in first approximation, the distributor offers the method `get_stock_info` to the brokers to fetch stock information. In reality, the distributor object does not *directly* provide that method. Instead, it offers a *factory* method, namely `provide_quoter_info`: invoking that method gives an instance of the `StockQuoter` class, which in turn offers the mentioned `get_stock_info`-method. These two methods are shown in Listing 1.4 and 1.5.

Listing 1.4. Quoter

```
interface StockQuoter
{
  StockInfo get_stock_info (in string stock_name) raises Invalid_Stock;
};
```

[1] The corresponding broker IDL3.x *component* is shown in Listing 1.14.
[2] See also the representation in IDL3.x in Listing 1.13 and 1.14 later.

Listing 1.5. Distributor (provide)

```
StockQuoter provide_quoter_info ();
```

Basically, the discussed methods cover the request and response interaction between distributor and broker. As discussed, to fetch stock information, the broker needs to get a stock-quoter from the distributor first; typically, it would store that information in a field, to query the stock quoters `provide_quoter_info` method to obtain the wished information. As accessor methods (getting/setting/unsetting), the broker supports three methods, shown in Listing 1.6

Listing 1.6. Broker

```
void        connect_quoter_info    (in StockQuoter c);    // set

StockQuoter disconnect_quoter_info ();                    // reset
StockQuoter get_connection_quoter_info ();                // get
```

2.2.2 Event-based communication Events are directed messages, from a sender to a receiver, i.e., generated by an event *source* and fielded by an event *sink.* In our example, the event source is part of the distributor, and the sink located at the broker. To receive notifications from a distributor, a broker needs to be *subscribed*, i.e., *subscription* is a relationship between a distributor and a broker.[3] To send events is also called *notification,* the source of an event *publishes* the event and a sink of an event *consumes* it. One also says, the event source *pushes* the event to the sink.

In the example, when the value of a stock changes, the distributor pushes the name of the stock to all subscribed brokers. The stock name is given basically as string (cf. the value type `StockName`). More interesting is the way, the event *notification* is done. The stock name is not just intended as a mere data type which is being exchanged. Instead, it is intended as a value that is exchanged by the *event mechanism* supported by the component model. This is also known as *event type*. So, the "string type" of Listing 1.11 should not be considered in isolation, but in combination with the *consumer* interface of Listing 1.9. An *event type* is represented in IDL2.x as a combination of a value type plus a corresponding *consumer interface*.

As mentioned, as broker needs to subscribe in order to be notified. The corresponding methods of the distributor are shown in Listing 1.7. The *subscribe* method returns an object of class `Cookie`. The return value is used to *identify* the communication between distributer and subscribed broker. In other words, the cookie is a unique *session identifier* and valid until the broker unsubscribes again. To unsubscribe, the broker hands over the id of the session it wishes to terminated and the corresponding method `unsubscribe_notifier` answers

[3] In IDL3.x, event communication is exchanged via the dual port types of *event sources* and *event sinks.* In the concrete case study, the two ports are named `notifier_out` and `notifier_in`. Cf. the IDL3.x component interface description of Listing 1.13 and 1.14 later. Note that in IDL3.x, the type of an event port (sink or source) is an *event type*. The event type `StockName` is given in Listing 1.15 for IDL3.x.

with the identiy of the broker, in case of a sucessful unsubscribing. The attempt to unsubscribe a non-existing subscription raises an exception and the cookie-argument is used to identify the subscription. Once a broker is subscribed, it will be notified about stocks whose values has changed in database.

Listing 1.7. Distributor

```
Cookie               subscribe_notifier (in StockNameConsumer c);
StockNameConsumer  unsubscribe_notifier (in Cookie ck)
  raises (Invalid_Subscription);;
```

When the value of a particular stock changes, the distributor uses the method `push_StockName` to notify all subscribed brokers. The broker does not directly provide that method; instead, it offers a factory method `get_comsumer_notifier`, which returns a `StockNameConsumer` object (cf. Listing 1.9), which in turn offers the mentioned `push_StockName` method. So the situation here is dual to the one for the request/response interaction, and especially the `StockNameConsumer` plays a role analogous to the one of `StockQuoter` for request and response (cf. Listing 1.4).

Listing 1.8. Broker

```
StockNameConsumer get_consumer_notifier_in ();
```

Listing 1.9. Stock name consumer

```
interface StockNameConsumer
{
  void push_StockName (in StockName stock_name);
  attribute Cookie cookie;
};
```

2.3 Data types

As mentioned, we present the *data types* for the values communicated in the system and the *exceptions*. As the interfaces, they are specified using IDL. We use records ("structs") and general data types ("value type") for the specification. We do not need the full generality of IDL for specifying the here (e.g., we do without inheritance and operations for the value types), and even without experience in IDL, the definitions should be fairly clear.

Listing 1.10. Stock info

```
struct StockInfo
{
  string name;     // full name
  long high;       // max value so far
  long low;        // min value so far
  long last;       // most recent
};
```

The information about a stock contains its name plus information about the latest value and some extremal values (`last`, `high`, and `low`). The stock itself is represented by its name; Listing 1.11 shows a wrapper class for the stock name.

Listing 1.11. Stock name

```
valuetype StockName
{
  public string name; // symbol of the stock
};
```

This value type is used in the communication between stock distributor -server and stock broker client to handle the callback from distributor server whenever the value of a stock the stock broker client is interested in changes. As mentioned in connection with the subscription of brokers at a distributor, a pair of a broker and a distributor are engaged in a *session*. The session id is represented by instances of the cookie-class, given in Listing 1.12.

Listing 1.12. Cookie

```
valuetype Cookie
{
  public string cookie_id;
};
```

Besides normal termination, some of the interface methods may also terminate by raising exceptions. The exceptions cover faulty situations when trying to unsubscribe when not eligible (i.e., basically, when not subscribed), or for a stock which does not exists. The two exceptions are `exception Invalid_Stock{}` and `exception Invalid_Subscription{}`.

2.4 IDL3.x

The following is taken from [4]. It models the quoter system using version 3 of the IDL. This allows a higher level of abstraction, in particular relying on the Corba component model. The system two main components, the distributor and the broker are shown in Figure 1, using the symbolic representation from IDL3.x, and their textual representation is given in Listing 1.13 and 1.14.[4].

Listing 1.13. Distributor

```
component StockDistributor supports Trigger{

  publishes StockName   notifier_out;       // event source
  provides  StockQuoter quoter_info_out;    // facet, provided interface

  attribute long        notification_rate;

};
```

The distributor plays the role of a publisher/server, and the stock broker the role of a subscriber/client.

Listing 1.14. Broker

```
component StockBroker {
   consumes Stockname notifier_in         // event sink
   uses     StockQuoter quoter_info_in    // receptacle, requ. interface
```

[4] Remember that in the previous representation in IDL2.x, `StockBroker` and `StockDistributor` only are *interfaces,* not components (cf. Listing 1.2 and 1.3)

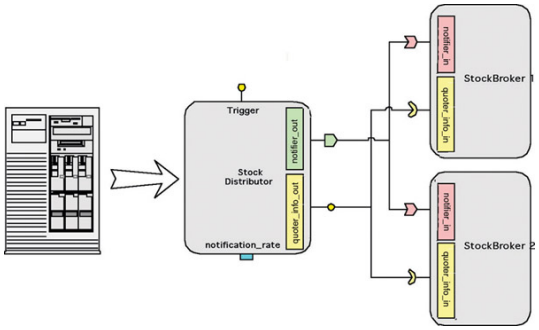

Fig. 1. The architecture of Stock Quoter System in IDL3.x

Listing 1.15. Stock name (IDL3.x)

```
eventtype StockName {
   public string name;
};
```

As in the case of the IDL2.x interface version earlier, the component here inherits the trigger interface (cf. Listing 1.1), offering methods to start and stop the component. The two components are dual to each other, reflecting their dual roles. The distributor publishes information that the brokers receive. More concretely, the two *ports* `notifier_out` and `quoter_info_out` of the distributor are mirrored by `notifier_in` and `quoter_info_in` of the broker. Apart from being dual to each other, the two components also illustrate two major patterns of communication within the component model. One based on *events* and the other by the request/response mechanism.

2.4.1 Event communication Component can communicate via events. More precisely, the distributor *publishes* the events via *port* `notifier_out` and the broker receives it via its dual port `notifier_in`. In order to receive notifications, the broker sink must be *subscribed*. This mode of communication is also called a publish and subscribe architecture.

2.4.2 Request/response The second mode of communication also uses ports. They are called *facets* and *receptacles*. The correspond to the concepts of *provided* and *required* interfaces. In the example, the interface in question is the `StockQuoter` interface (cf. Listing 1.4). It's the distributor that provides the interface, namely via the facet `quoter_info_out`. Dually, the broker requires that interface via the receptable `quoter_info_in`. Unlike in the situation for the event-based communication, the interface `StockQuoter` here is an "ordinary" method interface, not an interface for events. Concretely here, the `StockQuoter` mentions one method, which therefore is offered by the quoter and required by the broker. The bottom line is that the quote offers a method for clients to obtain information about a single name (identified by the name as a string).

The two entities `notifier_in` and `quoter_info_in` are *ports*. The `notifier_in` is an *event sink* So both ports are *input ports*, one is for *consuming*, the event sink for consuming the stock name event types, the other is called a *receptacle* Let's have a look at the broker's `quoter_info_in`-*receptacle*. The keyword, as said, to designate a receptacle is `uses`.

References

1. The common object request broker: Architecture and specification revision 2.6, May 2002. OMG Technical Document formal 05-09-02.
2. Corba components. OMG Document 2002-06-65, June 2002.
3. E. B. Johnsen, O. Owe, and I. C. Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theoretical Computer Science*, 365(1–2):23–66, Nov. 2006.
4. D. C. Schmidt and S. Vinoski. The CORBA component model: Part 2, defining components with the IDL 3.x types. *Dr. Dobb's Portal*, Apr. 2004.
5. T. N. Thuan, T. T. M. Thuong, T. V. Khanh, and N. V. Ha. Checking the consistency between UCM and PSM using graph method. In *Proceedings of ASCIID'09*, 2009.