A Notification Service for Next-Generation IT Systems in Air Traffic Control

C. Liebig, B. Boesling and A. Buchmann
Darmstadt University of Technology*

Abstract

Next generation Air Traffic Control (ATC) systems must be adaptable to different settings and flexible to handle new operational procedures in order to cope with the challenges of rapidly growing traffic. To achieve a high degree of maintainability, configurability and expandability, they should be constructed following the paradigm of component-oriented software architecture. Integration of components in such a distributed and heterogeneous environment has been specified by ATC authorities to be realized through the deployment of CORBA middleware. Whereas the main model of interaction in CORBA is one-to-one request-reply, many ATC applications show typical one-to-many interaction characteristics. In this paper we focus on applications that feature state change notifications, particularly operational data display services as specified in a recent OMG proposal. We present the design and implementation of our prototype for such a publish/subscribe notification service. Our prototype leverages upon TIBIOP, a CORBA addressing profile that implements GIOP on top of TIBCO's multicast messaging middleware.We discuss to which extent the multicast RMI features match the application requirements and discuss the resulting implications on service specification and implementation. We show how one can profit from subject-based addressing and that failure recovery can thereby be simplified.

1. Introduction

IT systems and applications in the ATC domain (in the following denoted by *ATC systems)* are complex systems by nature. ATC systems encompass many distributed and heterogeneous subsystems, operated by different organizations and deployed on a 24x7 basis. In most places technology from the 1970s and 80s is still in use - monolithic systems with poor maintainability, hard to expand with new functionality and to adapt to new requirements [28]. European ATC authorities identified the need to handle traffic that is expected to increase in excess of 100% by 2015 while operating flights more efficiently [9]. Not only does this require the introduction of more integrated and collaborative air traffic management concepts and procedures, but also requires ATC systems to be flexible to handle new operational procedures and adaptable to different settings. Therefore, a component-oriented software architecture is proposed for next-generation ATC systems. Integration of components and subsystems in such a distributed and heterogeneous environment is facilitated by the use of middleware [1,9] like CORBA [22] - a fact that has recently be recognized as a primary requirement by ATC authorities .

The standard interaction model in CORBA is remote method invocation (RMI), a client-server request-reply model which implies a one-to-one communication relationship and does not provide support for multicast. However, we can identify applications in the ATC domain which exhibit typical multicast characteristics. For example, the integration of flight data processing systems, radar tracking systems and weather information services feature one-to-many and many-to-many data dissemination. Another class of applications that have multicast properties is characterized by its use of state change notifications. Specifically, the controllers in the airport tower rely upon their HMI (human-machine interface) to monitor and guide on-ground, airborne and approaching aircraft, to read weather data, condition and use of runways, taxiways,

* In cooperation with Deutsche Flugsicherung (DFS), Offenbach

and to control the status of ILS (instrument landing system), beacons and similar technical facilities. In the envisaged modularized system architecture, state change notifications originate at the diverse backend IT systems that implement ATC business logic components and are propagated through a notification service to the HMI frontends at the controller's working position. In a recent OMG proposal [23] submitted by COMPAQ and Orthogon in cooperation with DFS, the Operational Data Display Service (ODS) is specified as a set of CORBA components and their interfaces, which provide a publish-subscribe notification service for such purposes. In this paper we will present a protoype ODS implementation and architecture on top of a multicast enabled middleware. We show how to exploit the subject based addressing scheme provided by the middleware and discuss simplifications to the architecture specified in ODS due to the availability of multicast RMI mechanisms.

The paper is organized as follows. In the next section we will present a scenario of the airport tower ATC system. We briefly describe the components of an event-driven workflow enactment system and illustrate the use of the ODS. In Section 3. we will introduce the ODS architecure as specified in the OMG proposal. We will then describe in Section 3.1. the TIB/Rendezvous multicast middleware and present the concepts of subject based adressing. We will also explain how multicast RMI features are integrated with CORBA. In Section 3.2. the architecture of our notification service prototype is introduced and compared to the ODS specification. First performance experiments and results are presented. We will discuss simplifications that are achieved through the use of multicast RMI and subject based addressing, with emphasis on failure handling in Section 4. Finally we discuss related work and conclude the paper with a summary of ongoing and future work.

2. Airport tower ATC system

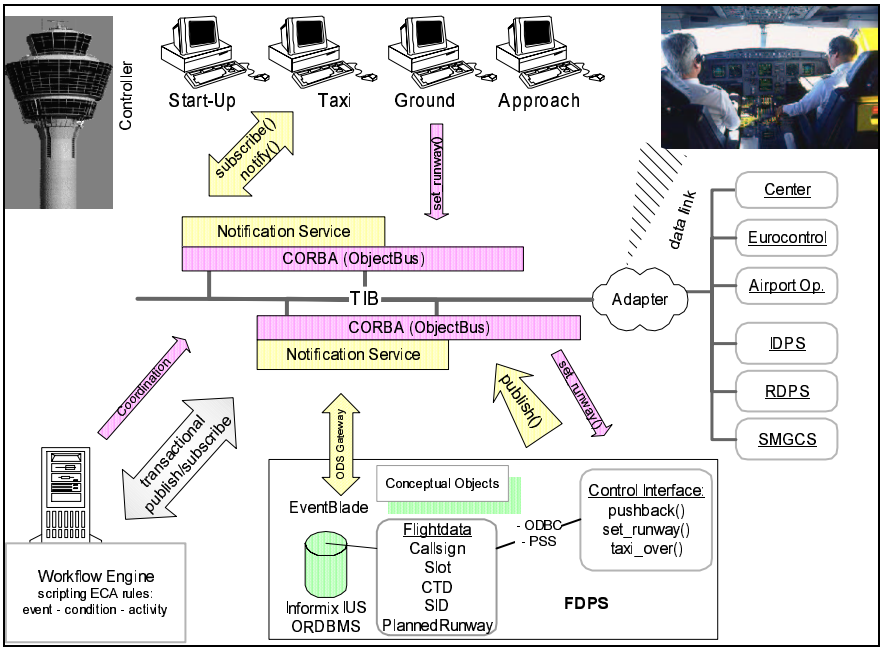
In order to handle the expected increase of air traffic, ATC must be highly efficient. As identified by German ATC authorities, the airports are likely to become the bottleneck. To enable the most efficient use of airport capacities next generation tower systems must well recognize the collaborative manner of operational procedures, support the controller in making the right decisions and exploit opportunities for automated processing while preserving correct and reliable operation. From an IT system architecture point of view, flexibility and extensibility are important requirements. Each of the 17 airports currently run by the DFS (Deutsche Flugsicherung) require different procedures imposed by the geographical layout of runways, taxiways, departure routes and organizational differences with respect to airport operators and facilities. Moreover, IT systems that are thought of being *external* today, e.g. airport operator facilities, airline IT systems, en-route ATC center, national and supra-national flight plan processing systems, will be tightly integrated with next-generation tower ATC systems [9]. The same holds for the aircraft on board computer which is connected by wireless communication networks, known as a *datalink*.

Therefore, we propose to apply a configurable event-driven workflow system architecture for the purpose of coordinating the various tasks and subtasks of ATC procedures for departure clearance and approach of aircraft. The tower ATC system does not autonomously take decisions other than to avoid accidents and is mainly supposed to provide the controller with current information for planning, scheduling and guiding aircraft movements. Additionally, the ATC system shall support the collaboration between controllers with interfering areas of competence, which correspond to specific geographical sections of the airport or airspace. Initially we focus on gate-to-runway control of aircraft and will deal with the airborne situations, especially radar data processing systems (RDPS), that typically have real-time constraints, in the next phase of the project. In fact, the most complex operational procedures with respect to the controller in the airport tower are involved with departure clearance and ground movement.

The departure clearance procedure typically involves several controllers with different roles. Initially positioned at the terminal block, the crew of an aircraft contacts the startup controller and negotiates the flight plan data, such as call sign, category of aircraft, slot, destination and standard instrument departure route (SID). Depending on the earliest slot time, current terminal block and taxi conditions, the startup controller eventually acknowledges the push-back request and hands over to the responsible taxi controller. Aircraft

must then be guided to the planned runway and lined up in such a way that they meet their preassigned slot times. In order to oversee the slot time constraint the tower system continuously estimates the so called *calculated time of departure* (CTD).The planned runway is initially chosen according to the standard instrument departure route (SID), which in turn depends on the flight destination, the runways in use, weather conditions and the time of day. Along with the general status information about weather, runways, taxiways, ILS, beacons etc., each controller monitors the flight plan data of the aircraft currently in his competence zone as well as flight plan data of aircraft in adjacent areas. Final line-up and take-off clearance is the responsibility of the ground controller. Additionally, collaboration is needed with approach control as take-off clearance for aircraft must be carefully coordinated with approaching aircraft. The approach controller lines up aircraft long before they enter the tower zone of competence. The ground controller then synchronizes take-off clearance with the approaching aircraft. Otherwise, the ground controller merely needs to monitor approaching aircraft until touch down. In a typical setting, approach controller and en-route controller are situated in a different organizational unit and location than the tower crew. Also, other vehicles and activities like buses, vans, fuel tanks, de-icing machines which are operated by different organizations and companies have to be monitored by the tower ATC system. Fig. 1. below shows the architecture of such a tower ATC system.

Fig. 1. ATC system architecture



.The main components are the HMI at the controller's working position, the Flight Data Processing System (FDPS) which is a database of all aircraft in and nearby the tower zone of competence, and the workflow enactment engine that controls the execution of the business logic based on ECA rules and transactional activities [7,8,11]. Additionally the system must be connected and integrated with various external systems like ATC center, weather information systems (IDPS), radar data processing systems (RDPS), surface guidance and control systems (SGCMS), logging facilities, and various other services.

At the heart of the event-driven workflow system lies the publish-subscribe notification service, which propagates work-flow events, i.e. begin/end/exception of activities, as well as application events, such as state change notifications for display purposes and triggering of reactions and follow-up activities. The state and business logic of 'real world entities', like flight plan data, taxiway, beacon etc., are encapsulated in CORBA components, which are called Conceptual Objects (CO) in the following. Each CO is represented by a CORBA object, and separated from the presentation logic following the Model View Controller concept [17]. The communication between the Model and Controller part is quite simple: they interact on a request basis. The communication between Model and View is realized by the ODS, a publish-subscribe notification service which by nature exhibits one-to-many communication relations.

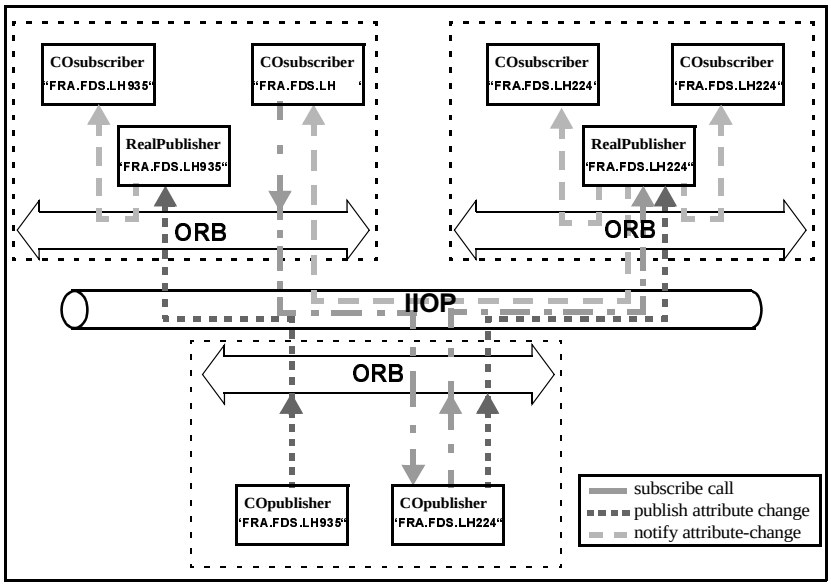
As a basic requirement, each HMI must provide the ability to display and access each CO managed by the tower ATC system. Note, that not every CO necessarily resides in the ATC system itself but may be exter-

nal, as well as there are external subscribers to CO's attribute changes which are managed by the ATC system. In the following section we will present more details of the ODS architecture as specified in the recent OMG proposal.

3. Operational Display Services Architecture and Implementation

The ODS architecture as specified in the OMG proposal is in essence a combination of observer and mediator pattern [10]. A `CO` publishes three types of events: *object creation*, *attribute change* and *object deletion*. The *object creation* event is passed to a central `COAdministrator` object which distributes this event to interested subscribers. Additionally, the `COAdministrator` creates one instance of a `RealPublisher` object, which acts as a mediator and is responsible to propagate *attribute change* and *object deletion* events to interested subscribers. Subscribers must register themselves at the `COAdministrator` for *object creation/deletion* events and directly at the `CO` for *attribute change* events. Furthermore, a subscriber may either register to receive all *object creation* events or only those which match a certain `CO` tag pattern. In the same way a subscriber may register at a `CO` for all *attribute change* events or only for changes of a specific set of attributes. The `COAdministrator` acts a s a proxy to `RealPublisher` factories and passes the object reference of the `RealPublisher` to the `CO` which then delegates all the work to the `RealPublisher`. Fig. 2. below depicts the architecture of the ODS - for simplicity we do not show the subscription and handling of object creation events.

Fig. 2. Architecture of the Notification Service



If a value of the `CO` is changed - e.g. through the controller interface of the `CO` in response to ATC controller action - the `CO` calls the respective method on its associated `RealPublisher` object which then notifies all interested subscribers about the occurrence of the event. In that manner the subscriber objects implementing the view part for the HMI may take appropriate display actions when state changes occur. The intention of differenciating between a `CO` and a `RealPublisher` is, that a `RealPublisher` shall be positioned close to the subscriber objects in order to limit the number of notification calls through the network. Furthermore, the `RealPublisher` acts as a mediator that keeps track of subscriptions, implements data dissemination logic and thus decouples the publisher from the subscribers.
Besides the complex failure and recovery scenarios imposed by the above architecture (which is dealt with in Sec. 4.) multicast semantic is implemented by sending one IIOP request message over the network to the `RealPublisher` which in turn sends out IIOP request messages to each possibly remote subscriber. In order to profit from a multicast enabled middleware, we changed and thereby simplified the notification service architecture. In the next section we will introduce the multicast features provided by TIB/Rendezvous and TIB/ObjectBus middleware and then present our prototype of the notification service.

3.1. Multicast enabled middleware

The multicast enabled middleware that we are evaluating as a platform for the tower ATC system, is based upon the notion of the *Information Bus* [27]. The concept of subject based addressing is similar to the idea

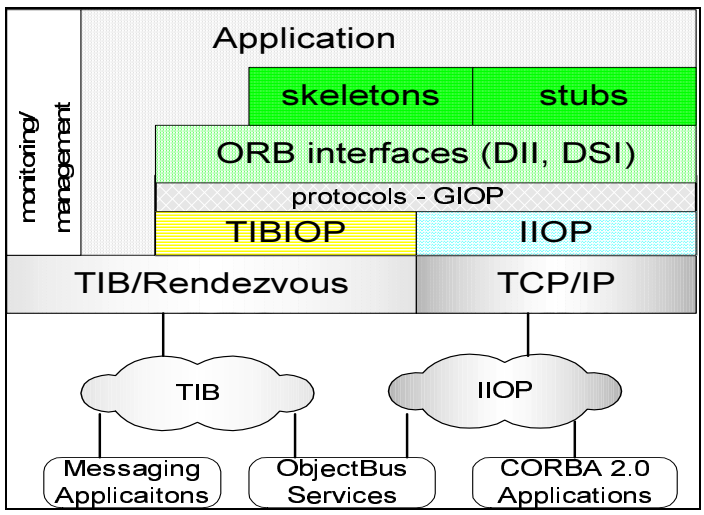
of a *tuple space*, first introduced in LINDA [4]. Instead of addressing a sender or recipient for a message by its identifier, which in the end comes down to a network address, messages are published under a subject name on the *Information Bus*, which is an abstraction provided by the TIB/Rendezvous middleware. The subject name is supposed to characterize the contents - i.e. the type - of a message. If a participant, who is connected to the *Information Bus*, is interested in some specific message types, he will subscribe for the subjects of interest and in turn be notified of messages published under the selected subject names. The subject name space is hierarchical and subscribers may use subject name patterns to denote a set of types to which he wants to subscribe (for an example see Section 3.2.).

Messages are constructed from typed fields, and can be recursively nested. Furthermore, messages are self-describing: a recipient of a message can inquire about the structure and type of message content. The abstraction of a bus inherently carries the semantic of many-to-many communications as there can be multiple publishers and subscribers for the same subject. The implementation of TIB/Rendezvous uses a light-weight multicast communication layer to distribute messages to all potential subscribers. On each machine, a daemon manages local subscribers, filters out relevant messages according to subject information and notifies individual subscribers. The programming style for listening applications is event-driven, i.e. eventaully the program must transfer control to the TIB/Rendzvous library which runs an event-loop. Following the Command pattern [7], the `onData()` method of an initially registered callback object will be invoked by the TIB/Rendezvous library when a message arrives with a subject that the subscriber is listening to.

Message propagation can be configured to use IP multicast or UDP broadcast. In the latter case, a special message routing daemon must be set up in each subnet in order to span LAN (broadcast) boundaries, comparable to the concept of tunneling in MBone [15]. Optionally, TIB/Rendezvous can make use of PGM, a reliable multicast transport on top of IP multicast, which recently has been developped by Cisco Systems in cooperation with TIBCO and proposed to the IETF [25].

Two quality of service levels are supported by TIB/Rendezvous: reliable and guaranteed. In both modes, messages are deliverd in FIFO order with respect to the publisher. There is no total ordering in case of multiple publishers on the same subject. Reliable delivery uses receiver-side NACKs and a sender-side in-memory ledger that buffers messages for some amount of time in case of retransmission requests. With guaranteed delivery, a subscriber may register with the publisher for a *certified session* or the publisher preregisters dedicated subscribers. Strict group membership semantics must be realized at the application level if so required. However, atomic message delivery is not provided. The TIB/Rendezvous library uses a persistent ledger in order to provide guaranteed delivery. Messages may be discarded from the persistent ledger as soon as all subscriber have explicitly acknowledged the receipt. In both variants, the retransmission of messages is receiver-initiated by sending NACKs.

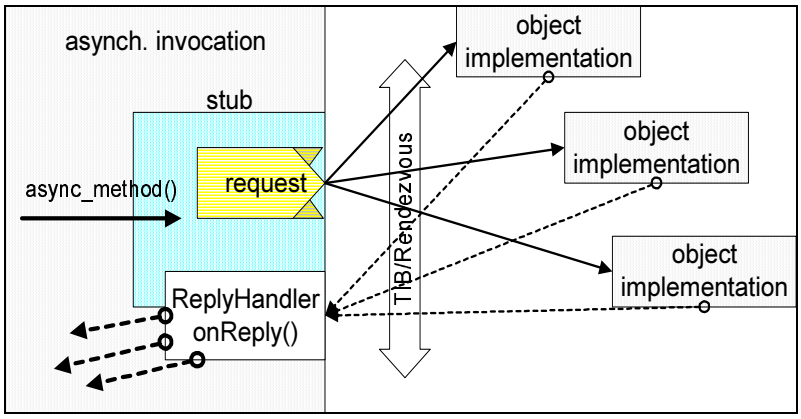
Fig. 3. ObjectBus Architecture



The implementation of our ODS prototype leverages TIBIOP, a registered CORBA 2.2 addressing profile on top of the TIB/Rendzvous layer [33]. The General Inter-ORB Protocol [15,22] defines the abstract message exchange protocol to implement remote method invocation. The standard inter-ORB protocol (IIOP) uses TCP/IP as a transport layer, in which case the Interoperable Object Reference (IOR) is based on the IIOP addressing profile, i.e. contains an IP host address and a port number (in addition to the object key). When using TIBIOP, the GIOP request messages are

marshalled into TIB/Rendezvous message and published on the *InformationBus* on behalf of a specific subject. The CORBA (server) object may be registered with the ORB presenting an application specific subject name. In that case the IOR is kind of the TIBIOP addressing profile and carries the subject name. In order to preserve interoperability, server objects may be registered with both, TIBIOP and IIOP profiles at the same time. On the other side, a CORBA client can construct a TIBIOP profile IOR from a subject name and let the ORB create a local proxy for it. When the client invokes a method on the proxy, the GIOP request message will be multicast through TIB/Rendezvous to all CORBA objects that registered with the same subject, introducing multicast RMI to the CORBA world. Two-way method semantic is supported using the mechanism of a `ReplyHandler`, as proposed in the CORBA Messaging specification for asynchronous method invocations [25].

Fig. 4. Asynchhonous multicast RMI



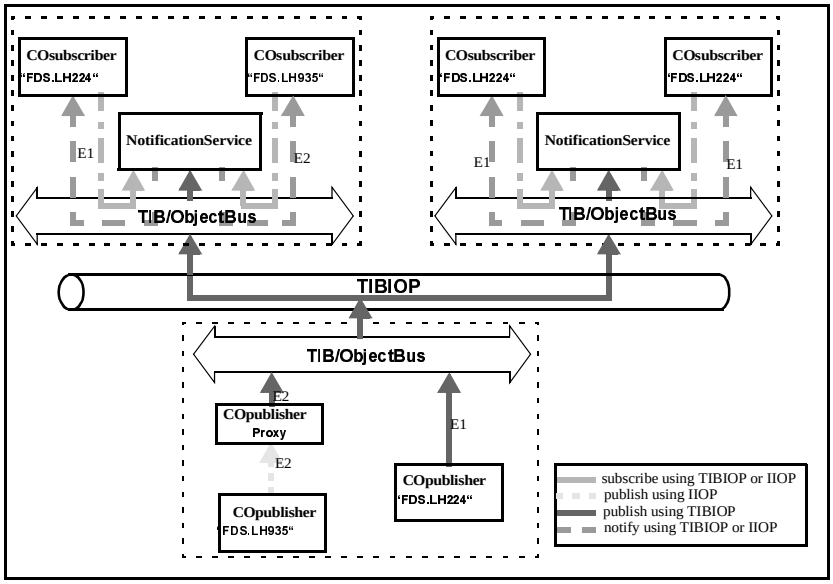
As depicted in Fig. 4., the multicast RMI may be executed at multiple sites, thus multiple return values and out parameters must be handled. The client invokes a so called *implied* stub method for the asynchronous version of the method call which takes as an additional parameter the `ReplyHandler` object. For each incoming result, a callback funtion is invoked to allow application specific processing of results. It is also possible to have synchronous multicast RMI invocations. The default is to process the first result, otherwise the application may provide a Collector in much the same way as the `ReplyHandler` is provided in the asynchronous case. Currently, guaranteed delivery is only provided with one-way methods, two-way requests are transferred using the reliable delivery mode of TIB/Rendezvous. Orthogonal to the different styles of method invocation and implementation ObjectBus provides a flexible threading model. In addition to the stub/skeleton and ORB APIs the application may access the TIB/Rendezvous messaging API directly.

3.2. Prototype Architecture and Implementation

Our ODS prototype encompasses three CORBA components: publisher objects, subscriber objects and the notification service. An instance of the notification service is located on every host where there are subscribers. The notification service internally comprises `RealPublisher` objects which relay incoming events from the `CO`s to the local subscribers, and a `Registration` object where a subscriber can register for events. Whereas the OMG Proposal designates that there is one `RealPublisher` designated per `CO`, we chose to have a `RealPublisher` for the `CO` on every host where there are interested subscribers. As registration is handled at the local notification service rather than directly at the CO, a subscriber can be started up time-independently from the publisher. On the other hand, the local notification service does not instantiate `RealPublisher` right away upon the *object creation* event of a `CO`, as in the case of ODS. Instead, the creation of `RealPublisher` objects is on demand, i.e. when the first subscriber on a host registers itself for some events of the respective `CO`.

Object creation events are handled collectively by a dedicated `RealPublisher` in the notification service component. Thus the HMI application can locally subscribe to *object creation* events and learn about temporary `CO`s which are dynamically created in the ATC system. In that way, we do not depend any more on a central `COAdministrator` object: `CO`s publish *object creation* events in the same way as all types of events, `CO`s need no more contact the COAdministrator, and subscribers register for object creation events in the same way they register for all other types of events.

Fig. 5. Notification Service Prototype



The basis of this architecture is the subject-based addressing scheme. We map the various event types and the object tags into the subject name space. Thus, depending on the event a publisher is going to disseminate, it generates a TIBIOP profiled IOR for a `RealPublisher` interface, parametrized with the corresponding subject. The publisher then invokes the respective method on the proxy derived from the IOR which results in a multicast RMI to all `RealPublisher` objects that have registered with the same subject-based addressing information. The `RealPublisher` on the other hand forwards the call to the locally registered subscribers. The IDL-Interface of the `RealPublisher` objects is shown below. The `set_xxx` methods correspond to *attribute change* events whereas the `obj_deleted` method signals the *object deletion* event. With `set_attributes` a change of multiple attributes is indicated.

```
interface RealPublisher {
      void set_long(in AttrName name, in long value);
      void set_float(in AttrName name, in float value);
      void set_string(in AttrName name, in string value);
      void set_object(in AttrName name , in Object value);
      void set_any(in AttrName name, in any value);
      //...
      void set_attributes(in AttrSeq attrs);
      void obj_deleted();
}
```

The subject namespace is defined as follows:

Event	Subject
single *attribute change*	ODS.<*domainname*>.modify.<*objtag*>.<*attrname*>
multiple *attribute change*	ODS.<*domainname*>.modify.<*objtag*>.special
object deletion	ODS.<*domainname*>.modify.<*objtag*>.delete
object creation	ODS.<*domainname*>.create.<*objtag*>

Every time a subscriber registers for events, the *RealPublisher* object complements his subject-based addressing information. At this point we make use of the possibility that an IOR can be composed of multiple profile components [15], i.e. the same implementation object is registered with the ORB under various subject names.

The following example illustrates the use of subject-based addressing. The code shows how a `RealPublisher` server object is created by the notification service, typically as a consequence of a local subscription. The `RealPublisher` is registered with the ORB using a TIBIOP addressing profile. The TIBIOP address is constructed from a subjectname pattern `"ODS.FRA.modify.FDS.LH224.>"`, suitable to receive all attribute value changes of a flight plan data set with object tag `"FDS.LH224"`.

```
OB_ReferenceGen *refGen;
refGen = new OB_ReferenceGen("IDL:ODS/RealPublisher:1.0");
```

```
refGen->append("ODS.FRA.modify.FDS.LH224.>","",1,(CORBA_Octet*)"");
CORBA_String_var iorString = refGen->toString();   // stringified IOR

CORBA_ImplementationDef_ptr implDefPtr;
impleDefPtr = new CORBA_ImplementationDef( iorString, (char *)NULL,
                                CORBA_ImplementationDef::SHARED_SERVER,
                                CORBA_ImplementationDef::REMOTE_ACTIVATION);

ODS_RealPublisher_ptr rp_ptr;        // out parameter for object reference
RealPublisher*  rp_obj = new RealPublisher();         // servant
OB_tie(*rp_obj, rp_ptr, implDefPtr, orb, boa, env);
```

First of all, the notifcation service creates an `OB_ReferenceGen` object, that encapsulates the server addressing profile for a `RealPublisher` interface and acts as a factory for stringified IORs.
In the next step, a TIBIOP addressing profile is appended with the subject-based address `"ODS.FRA.modify.FDS.LH224.>"` which matches all subjects beginning with the sequence `"ODS.FRA.modify.FDS.LH224"`.
After that, a CORBA implementation definition object is created that encapsulates the preceding information together with further parameters for the object adapter. The servant object for the `RealPublisher` is then created by instantiating the implementation class (`RealPublisher`), and registered with the ORB using the `OB_tie()` method of the object adapter. In general, it would be necessary to start the ORB event loop for dispatching invocations to the server-side object implementation. In our case the event loop has been initially started by the notification service and is already running.
As mentioned before, the `RealPublisher` implementation is part of the notification service component which encompasses the registration logic for local subscribers. When notifications are dispatched to the `RealPublisher` implementation, it will refer to the list of subscribers managed by the `Registration` object and forward the request appropriately.
On the publisher's side, when a `CO` needs to disseminate an attribute value change, the `CO` invokes a multicast RMI on the `RealPublisher` interface proxy. Assume for example, that for Lufthansa flight LH224 the calculated time of departure has changed because of delays due to de-icing. The `CO` creates an `OB_ReferenceGen` object encapsulating subject-based addressing information in accordance with the subject namespace layout. The stringified IOR is then used to create a local proxy which is narrowed to the RealPublisher interface. Now the CO may call a method, e.g. `set_string`, on that object reference. The appropriate code segment for the publisher `CO` is shown below.

```
OB_ReferenceGen *referenceGen;
referenceGen = new OB_ReferenceGen("IDL:ODS/RealPublisher:1.0");

referenceGen->append("ODS.FRA.modify.FDS.LH224.CTD","",1,(CORBA_Octet*)"");
CORBA_String_var iorString = referenceGen->toString(); // stringified IOR

CORBA_Object_var obj;
obj=orb->string_to_object(iorString,env);          // create proxy/stub

ODS_RealPublisher_ptr pub_ptr;
pub_ptr=ODS_RealPublisher::_narrow(obj);           // it is a RealPublisher

pub_ptr->set_string("CTD","13/1/1999 12:44", env);  // synchr. multicast RMI
```

The method invocation eventually results in publishing the TIBIOP request message under the subject `"ODS.FRA.modify.FDS.LH224.CTD"` through TIB/Rendezvous. If there are subscribers on remote hosts, the invocation will be dispatched to the respective `RealPublisher` and forwarded accordingly. For example, taxi controller and runway controller are interested in CTD. Additionally, external systems like ATC center, airline operator and destination airport might be interested in CTD changes in order to proactively adapt their schedules and planning.
The above coding example is somewhat simplified, as we make use of synchronous multicast. In that case a default `Collector` object is used by the proxy to gather results under the policy *first-one-wins*.

However, for the purpose of integrating the notification service with the workflow enactment service, we need more specific control on the results. Therefore, we actually must use asynchronous method invocation with specific `ReplyHandler` implementation in our prototype. This issue will be further discussed in a later section.

3.3. Performance Experiment

We conducted simple experiments in order to investigate the performance of multicast based RMI versus multiple unicast invocations. In a first experiment we measured the communication delay from the point of time that a publisher invokes a `set_string()` method to the point of time at which the method is dispatched by the ORB to the `RealPublisher` of the notification service. In a second experiment we measured the end-to-end communication delay to the point of time where the notification is delivered to a subscriber. We varied the number of hosts running a notification service from n=1..5, each with five registered subscribers. The publisher was then instrumented to run in two modes, one based on TIBIOP using multicast RMI and the second using multiple one-to-one RMI - in an asynchronous and multithreaded manner - to each configured notification service, again using TIBIOP.

The following table depicts the machine configurations that were used and their connection to the network

Host	Machine	Connection Bandwidth	Hops to Publisher
pub1	dual processor 300 Mhz SUN UltraSparc	100 Mbit	-
sub1	dual processor 50 Mhz SUN Sparc	100 MBit	1
sub2	dual processor 50 Mhz SUN Sparc	10 MBit	2
sub3, sub4	single processor 50 Mhz SUN Sparc	10 MBit	2
sub5	single processor 50 Mhz SUN Sparc	10 MBit	5

The publisher is running on host *pub1*, the notification service instances are running on hosts *sub1-sub5*. The hosts are connected to a switched 10/100 Mbit Ethernet, whereby *hops* counts the number of switches between publisher and subscriber. Hosts *pub1, sub1-sub4* are in the same IP subnet, *sub5* is in a different subnet, connected through an ATM backbone involving two ATM routers.The clocks were synchronized by NTP to the same peer, with a maximal offset below 0.8 ms. The basic network and machine load can be considered low, as we ran the experiments at after-office hours.

The table below shows the measurement results for average communication latency with respect to the `RealPublisher`. The numbers represent an average over 50 notifications, each separated by a distance of one second. The size of resulting TIBIOP request messages is about 487 bytes, when passing the timestamp of invocation as a string parameter. It can be seen that multicast RMI is slightly better than n-times asynchronous unicast. In both settings the average communication latency increases with the number of notification service instances attached. However, the multiple-unicast approach seems to be more sensitive with respect to addition of a lower bandwidth connected hosts, as is the case with sub5.

Subscriber Hosts	Multicast (ms)	Async. Unicast (ms)
sub1	8.52	9.34
sub1, sub2	7.88	8.41
sub1, sub2, sub3	8.8	10.62
sub1, sub2, sub3, sub4	10.4	12.5
sub1, sub2, sub3, sub4,sub5	11.84	13.6

In accordance with research done by A. Gokhale and D. Schmidt [12], the results of our experiments sup-

port the observation, that the main latencies are due to the time spend in stubs and skeletons for request marshalling and unmarshalling and that network latency is of minor impact. This observation becomes even more evident if we look at the end-to-end communication latencies, which additionally include the overhead of request forwarding from `RealPublisher` to the local subscriber processes. The table below shows the case for host *sub2* and multicast RMI, with one subscriber and five subscribers. Again, we incrementally added one to five hosts running a notification service:

Total # Subscribers Hosts	1 Local Subscriber at host sub1 (ms)	5 Local Subscribers at host sub1 (ms)
1	22.7	61.68
2	24.98	64.08
3	24.67	63.69
4	30.98	78.45
5	31.24	76.24

The experiments that we conducted so far do not consider situations where the ground load of the network is high, for example in case of multiple publisher running on different machines. Also, the settings are not large scale with respect to the number of hosts and the size of requests. This issues are investigated in our ongoing research.

4. Crash failure handling and reliability considerations

In this section, we will present the impact of ODS design on recovery from crash failures. We will then discuss the requirements with respect to reliable delivery of events.

4.1. Crash failure handling

The ODS specification discusses several failure scenarios and crash recovery procedures. The ODS specification considers the crash of the `COAdministrator`, crash of a `CO` and crash of HMI subscribers. However, it is not stated, what must be done in the case of failure of `RealPublisher` objects and what should happen if several of the participating objects fail. The situation is even more complicated, as communication failures reaching from large network delays up to network partitioning must be considered. It is beyond the scope of this paper to cover all scenarios which are relevant in a real-world setting. We will illustrate, how the design of our prototype notification service simplifies crash failure handling compared to the original ODS approach.
In order to provide basic availability qualities, the tower ATC system must be able to recover from crash failures of its components in a reasonable amount of time, which also means that recovery should be automatic as far as possible. Recovery from component failures is complicated in distributed object systems, as in addition to the state of object attributes communication relations must be re-established, introducing dependencies between components. The following table shows a “references” relationship, in terms of holding a CORBA object reference, for participating objects of the ODS:

	COpublisher	COsubscriber	RealPublisher	COAdministrator
COpublisher		-	X	X
COsubscriber	X		-	X
RealPublisher	X	X		-
COAdministrator	X	-	X	

Complex failure recovery scenarios exist because of the dependencies between objects residing on differ-

ent hosts. Moreover, the central `COAdministrator` instance marks a possible single point of failure. In contrast, the design of our prototype notification service avoids "references" relationships between remote objects by the use of the underlying multicast middleware. Therefore, crash recovery of the notification service is restricted to the locally affected components and thus preserves autonomy.

The ODS specification handles failure recovery of a crashed CO process, possibly containing several `CO` instances, as a deletion and reinstantiation of the crashed `CO`s. On restart, this requires registration with the central `COAdministrator` which replaces the old `RealPublishers` with new `RealPublishers`. It is further required that the `COsubscribers` register with the new `CO`s. We argue, that the crash of a CO should not be modeled as deletion and reinstantiation but as reincarnation of the same instance, with respect to the object tag of the `CO`. By leveraging the subject-based addressing scheme in our design, publishing of state change events is thus possible after restart without further actions.

When a HMI crashes and restarts it must resubscribe for the events it is interested in. With ODS, this requires to query the - remote - `COAdminstrator` for the object reference of the respective `CO`s. In our architecture, if the HMI saves its state in terms of subject patterns, the subscribers may directly resubscribe at the local notification service.

If the subscribers use TIBIOP, too, the recovery process can be automated. As the notification service may save the TIBIOP profiled IORs of its subscribers together with the subject patterns for subscribed events, the notification service can restore the object references and registration requests of the subscribers, and therefore the `RealPublisher` objects with the respective subject based addressing information can be reinstantiated. Note, that at recovery from failure a subscriber does not require a replay of former notifications, but will be reinitialized by an explicit poll of the state of the monitored CO or wait for state change notifications to arrive.

Another difficulty with ODS is the handling of `RealPublisher` failures. Although not discussed in the original specification, it can be easily seen from the above `references` relationship that besides the need to reregister HMI subscribers it is also needed to update the `COAdministrator` as well as the `CO` itself. The problem is complicated by the fact that all objects are remote with respect to each other and only the `RealPublisher` knows about enlisted subscribers. Thus either COs have to rely upon recovery initiated by the `RealPublisher` or subscribers must detect the failure and request failure handling at the `COAdminstrator`. Neither of the problems exist with our approach, as we use a distributed approach with independent `RealPublisher` objects on each host. If the HMI computer crashes, recovery may be implemented as described above. Even if processes are supposed to fail independently, it is still feasible to detect the failure of local notification service components and act appropriately.

4.2. Reliability considerations

As a first approach to distinguish the reliability requirements involved, we consider three coarse categories of events and notification service applications for tower ATC systems:

- Workflow Coordination Events
- Change notifications for directly collaborating parties
- Change notifications for monitoring parties

In the case of event-driven workflow enactment, activities are triggered by events and executed depending on the current state of the workflow [11]. Furthermore, activity related procedures are executed in a transactional manner on the basis of extended transaction models [8]. For example, begin and end of activities and their outcome as well as transaction events must be exactly once delivered, as those events are critical for the correctness of the workflow execution. Usually, there is no requirement for many-to-many communications and FIFO ordering is sufficient. In our current work this category of events is not distributed through the CORBA based prototype notification service. Instead we use the TIB/Rendezvous guaranteed messaging layer, with extensions for transactional publish/subscribe [5]. Additionally we need a group membership service, which can be realized at the application level using the preregistration of *certified sessions* supported by TIB/rendezvous in guaranteed delivery mode.

State change notifications for operational display purposes, as originally targeted by the ODS, require weaker guarantees, depending on the subscriber's role in the workflow execution. For example, if an aircraft leaves the competence zone of one controller, the aircraft must be handed over to the controller that is responsible next. The operational procedures are well defined in the workflow and basically, such a handover is modelled in the workflow as a state change. For example, when pushback is given, the aircraft is handed over from the startup controller to one of the taxi controllers. This may be visualized by the HMI as moving the corresponding flight plan data set from an *active listbox* to an *outgoing listbox* at the startup controller's HMI and from an *upcoming listbox* to the *active listbox* at the taxi controller's HMI.

Besides the directly involved controllers, there are further subscribers that merely monitor the proceedings of the respective flight without controlling it. In the latter case, the reliability of multicast RMI on top of TIB/Rendezvous reliable delivery mode are sufficient. As stated above, there is no need to persistently buffer events for later retransmissions, as only the most recent events are significant.

In the first case, however, the collaboration between the two controllers must be visualized in an *atomic* fashion. We argue, that it is still sufficient, to make use of the two-way RMI semantic and check in the `ReplyHandler` callback for successful delivery or otherwise catch exceptions. In fact, if a CORBA exception - or a timeout - is detected, then this situation must be dealt with by the workflow system and appropriate actions must be taken, as specified in the operational procedures. As a consequence, there is no need for stronger delivery guarantees at the multicast messaging layer. Similar arguments are presented in [6,29,34], where the authors state that providing atomic broadcast semantics like virtual synchrony at the message transport layer does not provide a practical solution for application level reliability requirements.

Still, the above approach has deficiencies with respect to scalability, as the two-way invocation semantic is specified at the interface level. As a consequence, all contacted servers for the same subject will respond with a GIOP message that is of no value for the application. This might lead to flooding of the publisher with useless responses and congest the network unnecessarily. We suggest to provide means to selectively set the method invocation semantic as a policy on a per server basis.

5. Related Work

Event-based computing is commonly recognized as an emerging paradigm for composing applications in open, heterogeneous distributed environments [2,3,11,18]. In CORBA, the Event Service [24] is introduced to provide a mechanism for decoupled, asynchronous event-style interaction between CORBA objects. The Event Channel acts as a mediator [10] between suppliers and consumers of events. It is distinguished between push- and pull-style interaction with the Event Channel. The Event Service interfaces may either be generic - using a CORBA `Any` event parameter - or interfaces may be typed using application specific types to transmit events. It is commonly recognized, that the Event Service has severe deficiencies [14] and in consequence the CORBA Notification Service [26] is proposed as a major extension, which additionally provides support for quality of service policy specifications and introduces mechanisms for event filtering based on a filter constraint language. One motivation for ODS, however, was to specify a lightweight notification service that particularly fits the needs of ATC systems and to avoid the complexity of a general constraint filter mechanism [23].

Orbix Talk [16] provides a publish/subscribe mechanism using so called *topics* in a similar way to subject based addressing. As with TIB/Rendezvous *topics* span a hierarchical name space. Each topic is mapped to a multicast group which is managed by a directory enquiry server. Subject patterns are not supported and publish operations must be defined as oneway. Like TIB/Rendezvous there are two quality of service levels, i.e. reliable delivery - using in memory message buffer - and guaranteed delivery in conjunction with the use of a persistent ledger.

Work on one-to-many communications in CORBA is mostly concerned with fault tolerance and replication [20,21]. The approaches are based on totally and causally ordered atomic broadcast, which is expensive to implement and does not provide the application with the adequate semantic [6,34].

D. Schmidt conducts comprehensive research on performance and real-time properties of CORBA, includ-

ing a real-time CORBA event service [12,14].

6. Conclusions and Future Work

In this paper we propose an event-driven component-oriented architecture for tower ATC systems and introduce a light-weight implementation for a notification service based on multicast RMI and subject based addressing. Using TIBIOP addressing profiles we can waive name services and mediator objects such as event channels, which may be a potential single point of failure or unnecessarily introduce complex interdependencies between remote components. Complex failure scenarios as described in the original ODS specification can be avoided. Therefore our approach presents a scalable solution with respect to reliability. We further expect that the multicast RMI based approach will exhibit better scalability in terms of performance, i.e. stability and steadiness [35], when the number of publishers and subscribers increases. For low load patterns, however, our first performance experiments show only marginal improvement of multicast over multiple one-to-one CORBA requests. In-depth performance tests are part of the ongoing work in our project. Further, we identified the need for a more flexible handling of multicast RMI invocation semantics, that would allow to invoke a multicast request in a mixed one-way and two-way manner, dependent on the semantic chosen by the server-side. We are also investigating, how to couple the notification service with a transactional workflow environment and in which way the strong reliability requirements for notification of workflow events can be incorporated. Another area of active research is the handling of disseminating radar tracking data for display purposes, which introduces additional real-time constraints.

7. Acknowledgements

We want to thank DFS for their cooperation. Special thanks to T. Heinlein for his support and the fruitful discussions on ATC systems.

8. References

[1] F. Barabas A. Poddany, J.-P. Florent and G. Klawitter. Java Shared Objects for Flexible Distributed Applications - Prototype of a Flight Data Management System. DIFODAM project, Eurocontrol, Brussels, http://www.eurocontrol.fr/projects/difodam/.

[2] D. Barret, L. Clarke, P. Tarr and A. Wise. A Framework for Event-based Software Integration, ACM Transactions on Software Engineering and Methodology, Vol. 5, No. 4, 1996.

[3] J. Bates, J. Bacon and K. Moody and M. Spiteri. Using Events for the Scalable Federation of Heterogeneous Components. In Proceedings of the SIGOPS European Workshop on Support for Composing Distributed Applications, September 1998.

[4] Carriero, Nicholas and D. Gelernter. Linda in Context. Communications of the ACM Vol 32 No 4, April 1989.

[5] A. Chan. Transactional Publish / Subscribe: The Proactive Multicast of Database Changes. ACM SIGMOD Conference, Seattle, Washington, 1998.

[6] D.R. Cheriton and D. Skeen. Understanding the Limitations of Causally and Totally Ordered Communication. In 14th ACM Symposium on Operating System Principles, Asheville, NC, December 1993.

[7] U. Dayal and A. Buchmann and D. McCarthy. Rules are Objects too: a knowledge model for an active, object-oriented database system. In Proceedings of the 2nd Intl. Workshop on Object-Oriented Database Systems, Lecture Notes in Computer Science 334, Springer, 1988.

[8] U. Dayal and M. Hsu and R. Ladin. Organizing Long-Running Activities with Triggers and Transactions. In Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data (SIGMOD'90), pp. 204-214, May 1990.

[9] Eurocontrol. EATMS Operational Concept Document, Ver. 1.1. Eurocontrol Brussels, http://www.euro-

control.be/projects/eatchip/ocd/
[10]E. Gamma, R. Helm, R. Johnson and J. Vlissides. Design Patterns. Addison-Wesley, 1994.
[11]A. Geppert and D. Tombros. Event-based Distributed Workflow Execution with EVE. In Proceedings of Middleware '98 (IFIP Intl. Conf. on Distributed Systems Platforms and Open Distributed Processing), September 1998.
[12]A. Gokhale and D.C. Schmidt. Measuring the Performance of Communication Middleware on High-Speed Networks. SIGCOMM Conference, ACM 1996.
[13]V. Hadzilacos and S. Toueg. Fault-Tolerant Broadcasts and Related problems. In S. Mullender, Editor, Distributed Systems, 2nd Ed., 1994
[14]T.H. Harrison, D.L. Levine and D.C. Schmidt.The design and performance of a realtime CORBA event service. In Proc. OOPSLA '97 Conference, October 1997.
[15]M. Henning. Binding, Migration, and Scalability in CORBA. Communications of the ACM, Vol. 41 No. 10, October 1998.
[16]IONA. OrbixTalk - The White Paper. Technical Report, IONA Technology, April 1996. http://www.iona.com/info/products/messaging/talk/whitepaper.html
[17]G.E. Krasner and S.T. Pope. A cookbook for using the model view controller user interface paradigm in Smalltalk-80. Journal of Object-Oriented Programming, 1(3), August/September 1988.
[18]B. Krishnamurthy and D.S. Rosenblum. Yeast: A General Purpose Event-Action System. IEEE Transactions on Software Engineering, Vol. 21, No. 10, October 1995.
[19]V. Kumar. MBone: Interactive Multimedia on the Internet. New Riders, 1996.
[20]S. Landis and S. Maffeis. Building Reliable Distributed Systems with CORBA. In Theory and Practice of Object Systems, John Wiley, April 1997.
[21]P. Narashimhan, L.E. Moser and P.M. Melliar-Smith. Replica consistency of CORBA objects in partitionable distributed systems. Distributed Systems Engineering, Vol. 4, No. 3., September 1997.
[22]Object Management Group. The Common Object Request Broker: Architecture and Specification, Revision 2.2. OMG, Famingham, MA, 1998.
[23]Object Management Group. Display Manager for Air Traffic Control, OMG Document: transprt/99-01-02, January 1999.
[24]Object Management Group (OMG). Event Service Specification. Technical Report formal/97-12-11, ftp://www.omg.org/pub/docs/formal/97-12-11.pdf.
[25]Object Management Group (OMG). CORBA Messaging. Technical Report orbos/98-05-05, ftp://www.omg.org/pub/docs/orbos/98-05-05.pdf.
[26]Object Management Group (OMG). Notification Service Specification. Technical Report telecom/98-06-15, ftp://www.omg.org/pub/docs/telecom/98-06-15.pdf.
[27]B. Oki, M. Pfluegl, A. Siegel and D. Skeen. The Information Bus - An Architecture for Extensible Distributed Systems. In 14th ACM Symposium on Operating System Principles, Asheville, NC, December 1993.
[28]T.S. Perry, In Search of the Future of Air Traffic Control, IEEE Spectrum, August 1997.
[29]J.H. Saltzer, D.P. Reed and D.D. Clark. End-To-End Arguments in System Design. ACM Transactions on Computer Systems, 2 (4), November 1984
[30]D.C. Schmidt and S. Vinoski. Overcoming Drawbacks in the OMG Events Service. SIGS C++ Report Magazine, June 1997.
[31]T. Speakman, D. Farinacci, S. Lin and A. Tweedly. PGM Reliable Transport Protocol Specification. Internet Draft <draft-speakman-pgm-spec-02.txt>, Cisco Systems, August 1998.
[32]Sun Microsystems. JavaBeans. http://java.sun.com/beans/
[33]TIBCO Inc. ObjectBus Whitepaper, TIBCO Inc., Palo Alto. http://www.ob.tibco.com/OB-white-paper.html
[34]W. Vogels, R. van Renesse and K. Birman. Six Misconceptions about Reliable Distributed Computing. ACM SIGOPS Europ. Workshop on Support for Composing Distributed Applications, Spetember 1998, Sintra, Portugal.

[35]P. Verissimo. Real-Time Communication. In S. Mullender, Editor, Distributed Systems, 2nd Ed., 1994