Master's thesis

Publish/subscribe
as architectural style for
component interaction

F. van de Laar

Supervisors:  dr. M.R.V. Chaudron
             ir. H. Maaskant
Advisor:      ir. C.J. Aarts

Eindhoven, August, 2002
Publish/subscribe

As architectural style for component interaction

F. van de Laar
Eindhoven 2002

Philips Research Laboratories Eindhoven
This report describes the results achieved during my nine-month graduation project at Philips Research Eindhoven. The project was done at the SWARCA-C cluster of the IPA group at Philips Research.

The exam committee consisted of:
Ir. C.J. Aarts (Philips)
Dr. M.R.V Chaudron (TUE)
Dr. C. Huizing (TUE)
Dr. J.J. Lukkien (TUE)

I would like to thank Michel Chaudron (TUE), Hugh Maaskant (Philips) and Christiene Aarts (Philips) for their guidance during my project. Furthermore, I would like to thank all of my colleagues at Philips for the pleasant working environment.

Frank van de Laar
Summary

This document describes the research into the use of publish/subscribe as an interaction style for software-component interaction. The research aims towards dynamic replacability of components, robustness of the system and the consequences of using publish/subscribe.

At first different variations of publish/subscribe were identified. From these the best suited ones were chosen for a first prototype – a text messaging system. The most important choice was the use of a broker that takes care of the message delivery to the subscribers. An initial set of interfaces was defined, together with interaction examples. These interfaces were used to implement the first prototype. This prototype consisted of publishers where messages could be typed in, and subscribers that displayed the messages. In this prototype, it was possible to dynamically replace a publisher.

After that an abstraction was made: Service Instances (SIs) were introduced. These consist of core code — which implements the functionality — and a set of publishers and subscribers — which are used by the core code to communicate with each other. Also a Configuration Manager (CM) was introduced. The CM takes care of the binding of SIs to corresponding brokers. Using these new building blocks a new prototype was created. This prototype also introduced robustness into the system. The CM detects SIs and brokers that have failed and restores the system. This was possible, because SIs without internal state were used. The second prototype made it also possible to dynamically replace a SI or a broker.

Finally, a third prototype was created that had to show the publish/subscribe system in a more real-life application. This application was chosen to be the control of a television. In Philips Research a simulation of TV hardware exists. Using this simulation, the prototype had to control the hardware, enabling it to tune to channels and implement a zoom SI. This zoom SI initially enabled the user to scale the image on the screen. This is useful on widescreen TVs when displaying 4:3 signals. A second zoom SI was created that automatically scales the image to fit the screen. The first SI could be dynamically replaced with the automatic zooming one.

Concluding, publish/subscribe as interaction style enables replacement of software components in running systems. Robustness can also be provided if stateless components are used. Most of the advantages of this publish/subscribe system are caused by the decoupling in space and time that is provided.
# Table of contents

Preface ........................................................................................................... i
Summary ........................................................................................................ ii

1 Introduction ............................................................................................... 1
   1.1 Scope ..................................................................................................... 1
   1.2 Assignment .......................................................................................... 1
   1.3 Report structure .................................................................................. 1
   1.4 Abbreviations ...................................................................................... 2

2 Variations in publish/subscribe ................................................................. 3
   2.1 Periodic vs. aperiodic .......................................................................... 3
   2.2 Push vs. pull ....................................................................................... 3
   2.3 Broker vs. no broker ........................................................................... 3
   2.4 Synchronous vs. asynchronous delivery ............................................ 4
   2.5 Topic- vs. Content- vs. Type-based .................................................... 4
   2.6 Filtered vs. unfiltered ......................................................................... 5
   2.7 One-to-many vs. many-to-many communication .................................. 5
   2.8 History vs. no history ......................................................................... 5
   2.9 Order preserving or non-deterministic order ....................................... 5
   2.10 Known vs. unknown origin ................................................................. 5
   2.11 Typed vs. untyped messages .............................................................. 6
   2.12 Replacement vs. queuing .................................................................. 6
   2.13 Priorities vs. no priorities ................................................................. 6
   2.14 Lossless vs. best-effort delivery ....................................................... 6

3 Phase 1: prototype on Windows 2000 ...................................................... 7
   3.1 Introduction .......................................................................................... 7
   3.2 Chosen publish/subscribe variations .................................................. 7
   3.3 Interfaces ............................................................................................ 7
      3.3.1 ISubscriber .................................................................................. 8
      3.3.2 IPublisher .................................................................................... 8
      3.3.3 IPublisherControl ....................................................................... 9
      3.3.4 ISubscriberControl ................................................................. 9
   3.4 COM components ............................................................................... 10
      3.4.1 Broker ....................................................................................... 10
      3.4.2 Publisher .................................................................................... 11
      3.4.3 Subscriber .................................................................................. 11
      3.4.4 BrokerlessPublisher ............................................................... 12
   3.5 Interaction between components ....................................................... 12
      3.5.1 Creation and destruction of a publisher and broker ...................... 12
      3.5.2 Destruction of the publisher and broker ....................................... 13
      3.5.3 Subscribing and unsubscribing .................................................... 13
      3.5.4 Publishing .................................................................................. 14
      3.5.5 Replacing a running publisher .................................................... 14
   3.6 Implementation ................................................................................... 15
      3.6.1 Broker ....................................................................................... 15
      3.6.2 Subscriber .................................................................................. 17
      3.6.3 Publisher .................................................................................... 18
   3.7 Conclusion .......................................................................................... 18
      3.7.1 Goals .......................................................................................... 18
      3.7.2 Specification .............................................................................. 19

4 Phase 2: Replaceability and robustness ................................................... 20
   4.1 Introduction ........................................................................................ 20
   4.2 Design choices ................................................................................... 20
   4.3 System overview ............................................................................... 21
4.3.1 Configuration Manager .................................................. 21
4.3.2 Service Instances ..................................................... 22
4.3.3 Brokers .................................................................. 22
4.4 Interfaces .................................................................. 23
  4.4.1 Previous interfaces .................................................. 23
  4.4.2 IHeartBeat .............................................................. 23
  4.4.3 IService ................................................................. 24
  4.4.4 IServiceInstanceControl ......................................... 24
4.5 Components ................................................................ 25
  4.5.1 Broker ................................................................... 25
  4.5.2 Generalized Service Instance ................................. 26
4.6 Sequence diagrams .................................................... 26
  4.6.1 Querying a service instance ................................. 26
  4.6.2 Initialization of the system .................................... 27
  4.6.3 Recovering a crashed broker ......................... 28
  4.6.4 Recovering a crashed SI ..................................... 29
4.7 Prototype .................................................................. 29
  4.7.1 Introduction ........................................................... 29
  4.7.2 Configuration Manager ......................................... 30
  4.7.3 Robustness ............................................................ 30
  4.7.4 Redundancy ........................................................... 30
5 Phase 3: Demonstrating the system ................................. 31
  5.1 Introduction ................................................................. 31
  5.2 TV simulator ................................................................. 31
  5.3 Prototype architecture .............................................. 32
    5.3.1 HW Control .......................................................... 33
    5.3.2 Remote Control .................................................... 33
  5.4 Design choices ........................................................... 34
    5.4.1 No global knowledge ........................................... 34
    5.4.2 Architecture without central broker .................... 34
    5.4.3 Introducing an AndBroker .................................... 34
    5.4.4 Horizontal communication between components .. 34
    5.4.5 One horizontal communication process simultaneously 35
    5.4.6 Replacing a control component ......................... 35
  5.5 Transfer of state .......................................................... 35
    5.5.1 Using the CM ....................................................... 36
    5.5.2 Using a separate channel .................................. 36
    5.5.3 Letting the SIs take care ................................... 36
    5.5.4 Issues ................................................................ 36
  5.6 Interfaces ................................................................. 36
    5.6.1 IHWControl .......................................................... 37
    5.6.2 IHWUser .............................................................. 37
    5.6.3 ISubscriberControl2 .......................................... 37
    5.6.4 IStateTransfer .................................................... 37
  5.7 Sequence diagrams .................................................... 38
6 Conclusions and future work .......................................... 40
  6.1 Replacing ................................................................. 40
  6.2 Consequences ........................................................... 40
  6.3 Fault-tolerance .......................................................... 41
  6.4 Future work ............................................................... 41
References ...................................................................... 43
Appendix A: MIDL interfaces ........................................... 44
List of figures

Figure 2.1: Push and pull communication ...................................................... 3
Figure 2.2 Communication models with broker .............................................. 4
Figure 3.1 Interfaces .............................................................................. 8
Figure 3.2 Broker component ................................................................. 10
Figure 3.3 Publisher component .............................................................. 11
Figure 3.4 Subscriber component ........................................................... 11
Figure 3.5 BrokerlessPublisher ............................................................... 12
Figure 3.6 Creation of publisher and broker ............................................... 13
Figure 3.7 Destruction of publisher and broker .......................................... 13
Figure 3.8 Subscribing and unsubscribing .................................................. 14
Figure 3.9 Publishing ........................................................................... 14
Figure 3.10 Replacement of publisher ....................................................... 15
Figure 3.11 Broker class diagram ............................................................. 17
Figure 3.12 Subscriber class diagram ...................................................... 18
Figure 3.13 Publisher class diagram ......................................................... 18
Figure 4.1 Example system ................................................................... 21
Figure 4.2 Example Service Instance ....................................................... 22
Figure 4.3 Interfaces ............................................................................. 23
Figure 4.4 Broker component ................................................................. 25
Figure 4.5 Generalized SI component ...................................................... 26
Figure 4.6 Querying a service instance ...................................................... 27
Figure 4.7 System initialization ............................................................... 28
Figure 4.8 A crashing broker is recovered .................................................. 29
Figure 4.9 A crashing SI is recovered ....................................................... 29
Figure 5.1 TV system overview ............................................................... 31
Figure 5.2 Broker per link ................................................................... 32
Figure 5.3 Central broker ................................................................... 33
Figure 5.4 Horizontal communication example ........................................ 38
Figure 5.5 Failure example ................................................................... 39
Figure 6.1 Publish/subscribe vs. Function-call .......................................... 40
Figure 6.2 Broker per node ................................................................... 41
1 Introduction

1.1 Scope

This report concludes the nine-month project and is written as a graduation report for the Eindhoven University of Technology (TUE) as well as a research report for Philips Research Laboratories Eindhoven. The project was done at Philips Research as part of the ITEA (Information Technology for European Advancement) Robocop project. The Robocop project’s goal is to define a component based software architecture for the middleware layer of high volume embedded appliances.

1.2 Assignment

The assignment is to investigate software component interaction according to a publish/subscribe mechanism. Particular points of interest are:

- Replaceability of a component in a running system
- Consequences on the interaction style provided by publish/subscribe
- Investigate the possibilities for fault-tolerance (robustness) of the system

The emphasis is more on design than on implementation.

This was translated in the following sub-tasks: the different variations of publish/subscribe mechanisms must be identified. It has to be examined which of these variations are best suited for this particular publish/subscribe mechanism. Furthermore, a prototype must be build. To demonstrate the replaceability, one component – that uses the publish/subscribe mechanism to communicate – will be replaced in a running system.

1.3 Report structure

Chapter 2 describes the publish/subscribe variations found throughout the literature, together with some extensions. Chapter 3 describes the design and implementation of the first prototype – a simple text messaging system. Chapter 4 explains the next step: a second prototype that provides dynamic replacement of software components and that introduces robustness to the system. Chapter 5 describes the final prototype: using the publish/subscribe mechanism to control a television simulation. Finally, Chapter 6 states the conclusions and future work.
### 1.4 Abbreviations

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CM</td>
<td>Configuration Manager</td>
</tr>
<tr>
<td>COM</td>
<td>Component Object Model</td>
</tr>
<tr>
<td>FBX</td>
<td>Featurebox</td>
</tr>
<tr>
<td>HIP</td>
<td>High-end Input Processor</td>
</tr>
<tr>
<td>HOP</td>
<td>High-end Output Processor</td>
</tr>
<tr>
<td>HorCom</td>
<td>Horizontal Communication</td>
</tr>
<tr>
<td>I²C</td>
<td>Inter-IC (Integrated Circuit)</td>
</tr>
<tr>
<td>ITEA</td>
<td>Information Technology for European Advancement</td>
</tr>
<tr>
<td>MIDL</td>
<td>Microsoft Interface Definition Language</td>
</tr>
<tr>
<td>QoS</td>
<td>Quality of Service</td>
</tr>
<tr>
<td>RC</td>
<td>Remote Control</td>
</tr>
<tr>
<td>SI</td>
<td>Service Instance</td>
</tr>
<tr>
<td>TUE</td>
<td>Eindhoven University of Technology</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
</tbody>
</table>
2 Variations in publish/subscribe

A publish/subscribe system, in general, consists of two types of parties: publishers and subscribers. A subscriber registers its interest in certain information. The publisher is the party that has the information and publishes this to its subscribers. In literature, several variations in publish/subscribe are described. This section lists the variations found in the literature (see References), together with some extensions.

2.1 Periodic vs. aperiodic

There are two possibilities [8] for the publisher when to send a message. Firstly, whenever an event occurs that could trigger an update. This situation is called aperiodic, because there is no known time between two messages. Secondly, the message can be send periodically, e.g., with a predefined schedule.

2.2 Push vs. pull

Both the subscriber and the publisher can initiate the data-transfer from publisher to the subscriber [1], [2]. In a publish/subscribe mechanism that uses push communication between the publisher and the subscriber, all data is 'pushed' by the publisher to the subscriber. That is, if a publisher has new data for the subscribers it sends the new data to the subscriber, i.e., the publishers initiates the data-transfer. In a pull system, the subscribers 'pull' the data from the publishers; this means that the subscriber initiates the data-transfer.

It is also possible to have a communication style somewhere between push and pull, e.g., if the publisher pushes only part of the data (notification) to the subscribers and the subscribers decide whether or not to pull all the data from the publisher as a result of that notification.

![Figure 2.1: Push and pull communication](image)

2.3 Broker vs. no broker

A broker is a software component that decouples [5], [10], [12] the publisher from the subscribers by taking care of the communication between them. Furthermore, the broker handles the subscriptions. In the literature, several other names are used: event channel [2], router [4], delivery manager [7], message broker [3] and event broker [9]. The publisher and the subscribers only communicate with the broker, which takes care of the further handling of the messages.

A broker provides decoupling (between the components) in space and time. Decoupling in time means that two communicating components do not have to be running at the same time. Decoupling in space means that two communicating components do not have to be aware with which component they are communicating.
When using a broker, there can be different communication styles (push and pull) for the publisher and the subscribers see Figure 2.2. For example, in the hybrid push-pull case (top-right picture) the publisher pushes the data to the broker, but the subscribers pull the data from the broker.

Every variation has a different place where the messages are retained:
- Push: messages are retained in the subscribers.
- Pull: messages are retained in the publisher and the broker
- Hybrid push-pull: messages are retained in the broker
- Hybrid pull-push: message are retained in the publisher

![Diagram](image)

**Figure 2.2 Communication models with broker**

### 2.4 Synchronous vs. asynchronous delivery

A message can be delivered to the subscribers in two ways: synchronous and asynchronous. When synchronous communication is used, the publishers send method finishes after all subscribers have received the data. If asynchronous delivery is used, the message is sent to the subscribers and the execution of the publisher continues immediately.

### 2.5 Topic- vs. Content- vs. Type-based

There are, basically, three types of publish/subscribe [5]: topic-based (also known as subject- or channel-based), content-based [11] and type-based [6].

Topic-based publish/subscribe allows subscribers to subscribe to message of certain topic. Content-based publish/subscribe continues on this and allows subscribers to further specify the messages they want to receive, e.g. by specifying a mask. Type-based publish/subscribe allows subscribers to subscribe to messages of certain types.

There could be a fourth type, where subscribers get all messages that are published in the system. However, this could be seen as a special case of any of the three above mentioned types.
2.6 Filtered vs. unfiltered

Filtering is closely related to the previous variation. The filtering of messages can occur at three places:
1. the broker (if there are brokers in the system)
2. the subscriber
3. the publisher (in the case when there are no brokers, because the subscribers subscribe to the publisher directly)

The idea of filtering of messages is that subscribers can subscribe to a certain kind of information. Therefore, filtering of messages in the broker or publisher provides content-based publish/subscribe.
Filtering can also be done at a higher level: using several brokers. This creates a topic-based publish/subscribe system, because every broker represents a different topic.

2.7 One-to-many vs. many-to-many communication

Some publish/subscribe systems can have only one publisher per topic (or only one publisher in the whole system); this situation is called one-to-many. If there are multiple publishers per topic (or system), this is called many-to-many [4], [8].
When using one-to-many communication, it is also possible that, at a given time, there are two publishers. However, only one is allowed to publish. This enables smoother runtime replacement of a publisher, because the to-be-replaced publisher does not have to be destroyed prior to the creation of the replacing publisher.

2.8 History vs. no history

Some applications might benefit from a history feature for new subscribers. If a new subscriber registers itself, it can request information about what messages were sent before it registered, thus obtaining information about the current state of the system. The number of messages that are kept can of course vary from one (the last message that was sent) to all.

2.9 Order preserving or non-deterministic order

This property specifies if the publish/subscribe system delivers the messages to the subscriber in the same order as they were send by the publisher(s) (order preserving) or if the message order is not maintained (non-deterministic order.) If message have priorities (see 2.13) priorities should be taken into account.

2.10 Known vs. unknown origin

It can be necessary for a subscriber to find out which publisher sent a particular message. The publish/subscribe system can provide a method for discovering that. ‘Normally’ the subscriber does not know which publisher sends the message; this is called decoupling in space [10].
2.11 Typed vs. untyped messages

Messages sent through the publish/subscribe system can be either typed or untyped. If untyped messages are used, the publisher and subscribers should agree (i.e. decide at design-time or run-time) on which type to use, so that they can convert the message to that type.

2.12 Replacement vs. queuing

If the broker – for some reason – is slow with the delivery of the messages, some new messages send by the server can obsolete a message that is still waiting (inside the broker) for processing. In that case, it is better to replace that obsolete message than to send it to the subscribers. On the other hand, if new messages do not obsolete older messages, the new messages can be queued instead.

2.13 Priorities vs. no priorities

Priorities can be applied to several different objects in a publish/subscribe system:
- individual messages, so that messages that are urgent are delivered before less urgent messages
- all messages from a certain publisher, so some publishers have priority over others
- subscribers, so that messages sent through the publish/subscribe system are delivered first to the subscriber with the highest priority
- all components, this is another way to give precedence to publishers and/or subscribers

2.14 Lossless vs. best-effort delivery

One aspect that plays a role in all communication styles is Quality of Service (QoS). The main issue in publish/subscribe systems is certainty of delivery of messages. There are two options: lossless delivery and best-effort delivery. The first means that the publish/subscribe system makes sure that all messages that are published by a publisher are delivered to all subscribers. Best-effort means that the publish/subscribe system will try to deliver all messages, but does not make sure they are actually delivered.
3 Phase 1: prototype on Windows 2000

3.1 Introduction

The first phase in the design of the publish/subscribe system was building a prototype on Windows 2000, in order to familiarize with the concepts of publish/subscribe as well as with Microsoft’s Component Object Model (COM, [13]), which should be used to implement the publish/subscribe system. However, the main goal is that is should be possible to replace a running publisher with another one. There did not have to be a complicated application using this publish/subscribe system. A simple text messaging system, with several subscribers that handle the message differently, would be sufficient. Therefore, it was chosen to make publishers that send text messages typed in by the user to subscribers that display the message they receive, in a different color.

3.2 Chosen publish/subscribe variations

This section lists what variations – as described in section 2 – were chosen for usage in the first step: the Windows 2000 prototype. The variations are in the same order as in section 2.

1. Aperiodic messaging: this is most natural for a text-messaging system, because the messages are not entered periodically.
2. Push: the assumption was made that all subscribers need all messages, this makes push the best option
3. Broker: decoupling in time is desirable; a broker is probably the best way to solve this
4. Asynchronous: from publisher(s) to broker and from broker to subscriber(s) is synchronous, the broker should make the communication from publisher(s) to subscriber(s) asynchronous
5. Topic-based: with only one broker (i.e. topic), because all subscribers need all messages
6. Unfiltered: because every subscriber must get every message that is published
7. Many-to-many: is the most flexible choice, this also allows for only one publisher and/or subscriber
8. No history: this is application specific and is not needed here
9. Order preserving: subscribers get the messages in the same order they were typed in
10. Unknown origin: known origin is not necessary for this simple application
11. Typed messages: all message are (text)strings
12. Queuing: because of the assumption that all subscribers get all messages, no message obsoletes an other message
13. No priorities: this is the easiest to implement
14. Best-effort: the delivery of messages is not critical (by choice)

3.3 Interfaces

In order to provide in all needed functionality, four COM interfaces are defined (see Figure 3.1). These are: ISubscriber, IPublisher, IPublisherControl and ISubscriberControl. In the following subsections the interfaces will be explained. With every interface, it is
listed which components - defined in Section 3.4 - call and implement the particular interface. In addition, the parameters are discussed.

Section 3.4 will define COM components providing these interfaces. Because all COM interfaces must inherit from IUnknown, this is not mentioned throughout the text; also the IUnknown methods (QueryInterface, AddRef and Release) are not listed in the method lists.

**Figure 3.1 Interfaces**

### 3.3.1 ISubscriber

The ISubscriber interface provides the functionality for receiving updates, i.e. for receiving published data. This requires only one method:

- **Update():** receive data from publisher or broker.

**Called by:** Publisher, Broker  
**Implemented by:** Broker, Subscriber

**Parameters:**

- **Update**  
  - **[in]** `char *` data: string containing the updated data  
  - **[in]** `short` length: indicates the length of the data string

### 3.3.2 IPublisher

The IPublisher interface provides all functionality that the outside world needs to start, stop and replace publishers. For those three tasks, three methods are defined:

- **Start():** notify the publisher that it can start publishing data  
- **Stop():** notify the publisher that is must stop publishing data  
- **SetDataSink():** tells the publisher to which ISubscriber interface it should publish its data

**Called by:** Broker  
**Implemented by:** Publisher

**Parameters:**

- **Start**  
  - no parameters
- **Stop**  
  - no parameters
- **SetDataSink**  
  - **[in]** `ISubscriber *` sink: pointer to the ISubscriber interface where the publisher should publish to
3.3.3 IPublisherControl

Basically, IPublisherControl provides the same functionality as ISubscriberControl, only from the publisher's viewpoint. It enables the client application to tell the broker that it starts or stops a publisher. In addition, it is possible to replace a running publisher. Therefore, these methods are defined:

- **Add()**: add a publisher to the channel
- **Remove()**: remove a publisher from the channel
- **Replace()**: replace a running publisher with another publisher; the replacement publisher is added to the channel implicitly.

The component that implements this interface should call the Start, Stop and SetDataSink methods of the publishers (which of course implements the IPublisher interface) in order to add, remove and replace publishers. The publisher identifier (pid) that is returned, is used to identify a publisher that must be replaced.

**Called by**: 'main' program  
**Implemented by**: *Broker*

**Parameters:**

- **Add**
  - [in] IPublisher * pub
  - [out] short * pid

- **Remove**
  - [in] short * pid

- **Replace**
  - [in/out] short * pid
  - [in] IPublisher * pub

3.3.4 ISubscriberControl

This is the interface that the subscriber uses for subscribing and unsubscribing itself to the topic. Because this is the only functionality that is needed, there are only two corresponding methods:

- **Subscribe()**: subscribe to the topic
- **Unsubscribe()**: unsubscribe from the topic

Note that these methods do not return an identifier. This is because the subscribers do not need to be identified (to be replaced.)

**Called by**: Subscriber  
**Implemented by**: *Broker*

**Parameters:**

- **Subscribe**
  - [in] ISubscriber * sub

- **UnSubscribe**
  - [in] ISubscriber * sub
3.4 COM components

For the next step, COM was used. COM components were defined that implement the above-defined interfaces and together form the publish/subscribe mechanism. The components are designed in such a way that it is also possible to create a publish/subscribe ‘connection’ without a broker, see subsection 3.4.4.

All components are implemented as an executable. Therefore, they all have a “main” procedure, i.e. a main thread of execution. This procedure is not listed in the figures.

3.4.1 Broker

The broker (see Figure 3.2) component is responsible for controlling the publishers and subscribers as well as providing a way for the publisher to publish its data. The first two responsibilities are (of course) established by implementing both ISubscriberControl and IPublisherControl. In addition, ISubscriber is implemented. This was chosen, because then it would be transparent to the publisher if it publishes to a broker or to a subscriber. Another advantage of this is that it is possible that brokers are connected to each other. Normally, the broker sends its messages to the ISubscriber interfaces implemented by subscribers, but because brokers also implement this interface, it is possible that a broker sends a message to another broker.

Because the ISubscriber and ISubscriberControl interfaces have to work close together, it was chosen to combine their implementations in one class: CBroker. This class takes care of relaying the messages to all subscribers.

The IPublisherControl interface is implemented in another class. It can be separated from the subscriber part (i.e. the CBroker class), because it does not need to have knowledge of the subscribers and the number of subscribers.

Figure 3.2 Broker component

Note that the broker implements three interfaces, but only two of them (both Controls) are exposed. This is because of the way the components interact. It will be made clear in section 3.5.
3.4.2 Publisher

The publisher component (Figure 3.3) is straightforward; it only implements the IPublisher interface. This is the publisher component for when a broker is also involved. In the case that there is no broker, the publisher must take care of more responsibilities. This will be discussed in section 3.4.4. The implementation of the IPublisher interface is provided by the CPublisher class.

![Publisher component]

Figure 3.3 Publisher component

3.4.3 Subscriber

The subscriber is also straightforward. It only implements the ISubscriber interface, in the CSubscriber class. Consequently, it provides all functionality that a subscriber should.

![Subscriber component]

Figure 3.4 Subscriber component
3.4.4 BrokerlessPublisher

The BrokerlessPublisher component is a fictional component, i.e. it will not be implemented, but is only here to illustrate how to use the defined interfaces to establish publish/subscribe without a broker.

Because there is no broker, the publisher must take over the ISubscriberControl role of the broker. Hence, the BrokerlessPublisher should also implement the ISubscriberControl interface. Another consequence is that the IPublisherControl interface is not needed.

Internally, the CBroker class is used for the distribution of the messages to the subscribers. The datasink for the publisher is set to the ISubscriber interface of the CBroker class, this obsoletes the SetDataSink method of the IPublisher interface. Because the broker is integrated in the publisher, it is not possible for multiple publishers to publish to the same topic – provided that subscribers can be subscribed to one topic at the same time. This disables the possibility to have a many-to-many communication style.

![Figure 3.5 BrokerlessPublisher](image)

3.5 Interaction between components

This section describes several common situations in terms of interaction diagrams. Each subsection illustrates one particular case. First, the starting and stopping of the publish/subscribe system are illustrated. After that, subscribing and unsubscribing is explained. Then, publishing of data is elucidated. Finally, it will be shown that the main goal of this phase – the replacing of a running publisher – is possible.

3.5.1 Creation and destruction of a publisher and broker

This subsection describes the creation of a publisher, and the connecting of the publisher to the broker. The procedure is depicted in Figure 3.6. The procedure for when a subscriber creates a broker is very similar to this procedure; it is done through a COM’s CoCreateInstance call as well. Therefore, it will not be described in this section.

1. The main program calls the new method for the creation of a publisher.
2. After that, it calls COM’s CoCreateInstance, to create (or obtain a pointer to) a broker.
3. Then main calls the brokers Add method, to add the newly created publisher, this call returns a unique identifier for the publisher.
4. The broker sets its ISubscriber interface as datasink for the publisher and calls the publisher’s Start method.
3.5.2 Destruction of the publisher and broker

This subsection describes how a publisher is deleted, in the case that this publisher holds the last reference to the broker. Thus, the broker is destructed as well.

1. The main program wants to remove the publisher and calls the broker’s Remove method.
2. The broker first calls the Stop method, to stop the publisher from publishing data.
3. Then, the broker sets the datasink to NULL.
4. Because the reference count to the broker reaches 0, it is destructed.
5. The same goes for the broker; it is destructed as well.

3.5.3 Subscribing and unsubscribing

The subscribing and unsubscribing procedures are uncomplicated:

1. The subscriber creates (or obtains a pointer to) a broker, and queries for the ISubscriberControl interface.
2. It calls the Subscribe method, to subscribe itself to the published messages.
3. When the subscriber wants to stop the reception of updates, it calls the broker’s Unsubscribe method.
3.5.4 Publishing

Suppose there is a running system with two publishers and two subscribers. The following figure depicts the situations where each one of the publishers publishes a message. Note that this description fits both publishers.

1. After some processing, the publisher calls the broker’s Update method. This call returns immediately, because the message is buffered in the broker.
2. The broker distributes the message to both subscribers.

If the second publisher sends its message while the broker is busy distributing publisher 1’s message, publisher 2’s message is distributed to the subscribers after the distributing of the first message has completed. This establishes the order preserving property of the system (see Section 3.2).

![Publishing Diagram](image)

**Figure 3.9 Publishing**

3.5.5 Replacing a running publisher

An important feature of the system is that it is possible to replace a running publisher with another one. The initiative for this lies with the main program. It can create a new publisher and call the broker’s Replace method to stop the running publisher and let the new publisher start. In this case – the messages are simple text strings – this is not very useful, but in some cases it can be very useful, e.g. when a new publisher with improved quality or functionality is available.

Figure 3.10 illustrates the replacement of a running publisher.

1. The system is running and Publisher 1 is publishing to the broker.
2. The main program takes the initiative for the replacement of Publisher 1, and creates a new publisher.
3. After that, it calls the broker’s Replace method with Publisher 1’s id and the pointer to the new Publisher, Publisher 2.
4. The broker stops publisher 1.
5. The datasink for Publisher 1 is set to NULL again
6. The broker sets the datasink for Publisher 2.
7. After that, Publisher 2 is started.
8. Publisher 2 is publishing to the broker.
3.6 Implementation

This section describes the implementation details of the publish/subscribe system. As in the previous sections, the COM methods are not listed. In addition, the constructors and destructors are not mentioned. More detailed knowledge of COM might be needed to read this chapter. Because the emphasis is more on the design than the exact implementation, this chapter may be skipped.

3.6.1 Broker

The class diagram showed in Figure 3.11 shows the implementation architecture of the broker. The actual broker component (CBroker) is implemented in one class and not in two different classes like described in Section 3.4.1. However, more classes are created for the broker. This is because the broker is a COM out-of-process server. It therefore needs a so-called classfactory that takes care of instantiating CBroker classes. Also, a CServer class is created to implement the actual server; this class starts the classfactory and registers it with COM. In addition, because the Broker is an executable, there is a WinMain procedure that takes care of registering and unregistering the server as well as instantiating the CServer class.

CServer
The Open- and CloseFactory methods (called by WinMain) start or stop the classfactory. They also register or unregister the classfactory with COM. ObjectsUp and ObjectsDown, increase or decrease the server’s reference count (m_cObjects). CServer halts its execution if this count reaches 0; it then posts a quit message to the WinMain thread (identified by the m_dwThreadId attribute.) The CServer class has two more attributes: m_pCFBroker, which is a pointer to the classfactory and m_dwCFBroker, which is an identifier used to unregister the classfactory with COM.

CFBroker
This class is the classfactory for the CBroker class. Basically, all it does is create an instantiation of the CBroker class. Because all publishers and subscribers should use the same broker, CFBroker keeps a pointer to the CBroker that it created. If any successive requests for a broker take place, that same pointer is returned. The CreateInstance method is called by COM’s CoCreateInstance method, to request a pointer to a CBroker object. The LockServer method is not implemented, because it was not needed.

CBroker
The CBroker class performs the actual broker tasks. Therefore, it only implements the methods as described in Section 3.3. CBroker maintains two lists: a list of connected publishers and a list of connected subscribers. It also has a queue with messages implemented by the CCharFifo class (see below.) In addition to the main thread, CBroker
has another thread (MsgThread.) Just as the name suggests, this thread takes care of the
sending of the messages to the subscribers. Using a thread provides the required (see
Section 3.2) asynchronicity.

CCharFifo
This class implements a FIFO buffer (queue) by creating and maintaining a linked list of
ITEMs. An ITEM is a struct containing a pointer to the next ITEM, a char array for the
actual content and a boolean for indicating a quit message. This quit message is used for
stopping the broker when both the publisher- and the subscriberlist are empty. CCharFifo
uses a mutex to ascertain exclusive access to the list. The head and tail attributes
respectively point to the head and the tail of the list, these are needed to get the first
message and append a new message.
The IsEmpty methods returns if the buffer is empty. The Get method retrieves the first
message of the buffer while Put appends a new message to the buffer. Both Get and Put
use – in addition to the mutex – a semaphore to indicate the presence of messages. Put
performs a V (in Win32: ReleaseSemaphore) operation when a new message arrives.
Likewise, Get performs a P operation (WaitForSingleObject) before returning the
message. Thus, the Get method is blocking. The only object that calls the Get method is
the message thread. That poses no problem, because it should only be active when there
are messages in the buffer.

UniqueId
The UniqueID class provides the unique identifiers needed for the implementation of the
IPublisherControl interface. The identifier returned by GetID is in fact the first available
element of an array of bits, implemented through a bitfield. The RemoveId method frees
the specified identifier. The List class uses UniqueID for uniquely identifying publishers.

List
The List class implements a list. The actual implementation is an array – of a struct
named UNKNOWNS – that is dynamically enlarged or reduced, if needed. The
UNKNOWNS struct consists of an IUnknown pointer and a publisher id (only used for
the list of publishers, not for the list of subscribers). The methods Add and Remove
respectively add and remove an item from the list. Both the Add method and the Remove
method are overloaded to provide support for both the publisher- and the subscriberlist.
3.6.2 Subscriber

Contrary to the broker component, the subscriber only consists of one class (CSubscriber), shown in Figure 3.12. The subscriber component uses two threads. The main thread waits for the user to indicate that the subscriber should stop. The other thread (ThreadFunc) is the actual worker thread: it instantiates the CSubscriber class, and takes care of registering it with the broker.

CSubscriber

This time the constructor is explicitly shown, because it is not standard: there is a parameter color. This is only used by the application creating the subscriber, for selecting color for the text displayed when the Update method is called. The Update method is the only method in the CSubscriber class.
The attribute m_cRefs is the reference count for the subscriber object. Like the broker, the object is deleted when this count reaches 0. The m_hStdOut handle is used for setting and restoring the text color. The third attribute is used to store the text color, prior to the setting of the selected color. This way, it can be restored.

![Subscriber class diagram](image)

**Figure 3.12 Subscriber class diagram**

### 3.6.3 Publisher

The last component is the publisher (Figure 3.13) Again, this component consists only of one class (CPublisher.) This class is instantiated by the application that wants to use a publisher for communication. In this case, there is a Main procedure that waits for text input and then calls CPublisher's Publish method. This method is not a method inherited from IPublisher, but it is added to provide a way for the user of the CPublisher class to send data through it.

**CPublisher**
The Start and Stop methods are implemented through the use of the m_bStopped boolean. If this boolean is set to true, the Publish method does not call the broker, but discards the information. The initial value is true; when the SetDataSink method is called, it is set to false. The attribute m_cRefs is again the reference count for the object. m_pSink contains the pointer to the sink that is set by the broker using the SetDataSink method.

![Publisher class diagram](image)

**Figure 3.13 Publisher class diagram**

### 3.7 Conclusion

#### 3.7.1 Goals

The first goal was to familiarize with the concept of publish/subscribe and COM. The second one was to be able to replace a publisher with another one.

The first goal is met. Learning and understanding COM did take a lot of time, but eventually everything was clear.
It is possible to replace a publisher, by using the returned identifier together with the broker's Replace method. Now, the replace is implemented by successive calls to Add (for the new publishers) and Remove (for the to-be-replaced publisher.) First setting the datasink for the new publisher and then calling Stop (old publisher) and Start (new publisher) can make this smoother.

3.7.2 Specification

Section 3.2 specified which variations of publish/subscribe should be implemented in this first prototype. The following text motivates why these specifications were met. The numbers correspond to the number in section 3.2.

1. _Aperiodic messaging_ is achieved, because the publishers publish data every time the user enters a message.

2. All communication takes place through the _push_ way, the publisher calls the broker's update method. This causes the broker to call the update method at all connected subscribers.

3. A _broker_ is used.

4. The _asynchronicity_ of the delivery of the message from the publisher to the subscribers is realized, because of the thread that dispatches the messages to all subscribers. The calls from publisher to broker and from broker to subscribers are all synchronous, the thread makes the total delivery asynchronous.

5. _Topic-based communication_ is accomplished by the introduction of one broker (i.e. one topic). All communication takes place through that broker.

6. Message are not being filtered in the broker, subscribers get all messages, thus: _no filtering_.

7. Multiple publisher and subscribers can use the broker simultaneously, thus _many-to-many_ communication is possible.

8. The subscribers cannot request previous messages from the broker, therefore there is _no history_.

9. The messages are (when received by the broker) put into a _first-in-first-out_ (FIFO) buffer. This, together with the fact that every message is send to all subscribers prior to the getting of the next message, ensures that the messages are delivered in the same order as they were send by the publishers.

10. The subscriber does not know, and has no way to find out, which publisher a received message origins from, thus _unknown origin_.

11. The messages are of one type, namely char *, so they are _typed_.

12. Because of the FIFO buffer, messages cannot be replaced with an updated message. Therefore, _queuing_ is used.

13. _No priorities_ are explicitly set. All threads use the default priority. The messages are delivered in FIFO order.

14. If the delivery of a message fails, the broker does not reattempt the delivery. Therefore, the delivery is _best-effort_.

Phase 1: prototype on Windows 2000
4 Phase 2: Replaceability and robustness

4.1 Introduction

The primary goal of this phase is to show that using a loosely coupled communication paradigm like publish/subscribe provides easy replacement of parts of the system. There are four secondary goals defined for the second phase:

1. Use the previous defined interfaces at a higher level, creating so-called Service Instances (SIs) (defined in the next section).
2. Introduce a Configuration Manager (CM) that takes care of the binding of the SIs and the brokers.
3. Use the CM to replace running SIs and brokers
4. Have a robust system, i.e. a crashing component or broker does not cause a crash of the whole system.

This phase is also discussed in [1][12][12].

First, the architecture of the system will be discussed. After that, the will be more detail on the interfaces, components and interactions between the components that form the system. The final section will describe a prototype that implemented this system.

4.2 Design choices

The system architecture – as described in the next sections – is based on several design decisions. Some of them are already mentioned in the introduction section.

Introducing Service Instances
Service Instances are introduced, because this makes it possible to design the system at a higher level. Not just a simple publisher or subscriber, but a collection of publishers and subscribers that are used by the SI to communicate with its environment.

On-demand creation of brokers
There are two possibilities for the moment in time when a broker is started.

1. When either a publisher or subscriber - for the topic that the broker is used for – is present to the system. A variation on this is that a broker is started when a publisher is present. The broker can then save the messages published by that publisher until a subscriber is present.
2. When both a publisher and a subscriber are present in the system. This is the best option for saving resources, because unnecessary broker creation is avoided. However, this can not be applied in the case that the broker must save published messages.

The prototype (see Section 4.6) uses the second option, but the general design allows for both options to be implemented.

Third-party binding
A choice had to be made between first-party binding – in which the SIs bind themselves to brokers – and third-party binding – in which a third party takes care of the binding. Third-party binding was chosen for two reasons:

1. It allows both broker creation options.
2. It decreases the dependencies between SIs and brokers.

The Configuration Manager takes care of the binding.

Heartbeat
For robustness, a so-called heartbeat was introduced. The CM periodically checks if all parts of the system are still running. If a part has failed, the CM can restart it. The third-
part binding choice, simplifies this restarting for the CM, because the CM is aware of the 
system architecture (i.e. it knows to which parts of the system are bound to the part that 
has failed).

**No state**
SIs do not have an internal state. This was chosen to circumvent certain consistency 
problems when replacing SIs in a running system. Replacing SIs with internal state 
require the transfer of state between the old SI and the new one. State transfer will be 
shortly discussed in the next phase, see Chapter 5.

4.3 System overview

Figure 4.1 shows an example system: one CM, three SIs and three brokers – one for each 
communication channel (topic). The solid arrows indicate the flow of messages in the 
system, whereas the dotted arrows indicate control. The CM controls all SIs and brokers. 
This is because the CM takes care of the binding of the SIs to the brokers. The SIs use the 
brokers to communicate with each other.

![Diagram](image)

**Figure 4.1 Example system**

4.3.1 Configuration Manager

The Configuration Manager has the following tasks:
- Instantiate SIs
- Instantiate brokers, if needed
- Bind the SIs to the brokers
- Maintain the integrity of the system

Note that the CM contains all information about the system, this is needed for 
maintaining the integrity of the system (see below). This makes the CM a single point of 
failure. It is assumed that the CM is unfaultable.

**Instantiate SIs**
The CM is responsible for starting the SIs. There are two options for when to do this:
1. The CM knows when it is started which SIs need to be started to obtain the desired 
system configuration.
2. The CM provides a way for an external party (e.g. a user) to select the SIs that should be started. The SIs are not started until this is indicated by the external party.

**Instantiate brokers**

After starting a SI, the CM checks that SI, which channels it, needs for publishing and subscribing. That information is combined with existing knowledge of the system. On the basis of this information, the CM decides whether or not to start a broker.

**Bind of the SIs to the brokers**

If the CM has started a broker, it binds all SIs that wish to use the channel that the broker was created for, to this broker. This is called third-party binding.

**Maintaining the integrity of the system**

In order to keep the system running, the CM periodically checks all SIs and brokers for their existence. If a broker or SI does not exist anymore, the CM restarts it and restores the bindings to this SI/broker. Brokers also have some robustness code inside. If an Update to a subscriber fails, the broker removes that subscriber from its list. It assumes that the corresponding SI has crashed and that the CM will restart and rebind that SI.

### 4.3.2 Service Instances

Service Instances make up the actual system. The CM and the brokers provide the SIs with robustness and means of communication. Figure 4.2 shows an example of a SI.

SIs consist of core code – which implements their actual functionality – and both publishers and subscribers for communication with its environment.

The meaning of the Conf. and HB interfaces will be explained in Section 4.4.26.

![Figure 4.2 Example Service Instance](image)

### 4.3.3 Brokers

The brokers take care of relaying messages from publishing SIs to receiving SIs. They also play an important role in the replacing: brokers decrease the number of SIs that should be informed about the replacing of a SI. For example, imagine a system with one SI that publishes and several SIs that subscribe to one topic (broker). If the publishing SI is to-be-replaced, there is no party that should be informed about this, because the broker does not know its publishers. However, if no broker would be used (i.e. all subscribing SIs are connected to the publishing SI), all subscribing SIs should be informed that they should unsubscribe. This advantage is caused by the decoupling in space provided by the broker. This is also good for robustness: the CM does not have to reconnect all SIs that are communicating with a crashed SI, but only one a few (all connections from the crashed SI to the broker it is using).

Another advantage is that brokers enable multiple publishers to the same topic. If the publisher also would take the broker role, there cannot be more publishers (under the
assumptions that a subscriber can connect to only one publisher/broker). If this is desired the SI should implement several subscribers for the same topic.

4.4 Interfaces

This section first describes and motivates the changes in the interfaces as defined in chapter 3. The interfaces will be treated in the same order as in Chapter 3. After that, the new interfaces for this phase are defined. Figure 4.3 shows an overview of the interfaces used in this phase. Appendix A lists all interfaces for this phase in Microsoft’s Interface Definition Language (MIDL). This definition was used for the prototype.

<table>
<thead>
<tr>
<th>IPublisher</th>
<th>IService</th>
<th>ISubscriber</th>
</tr>
</thead>
<tbody>
<tr>
<td>+Start(); HRESULT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+Stop(); HRESULT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+SetDataSink(sink : ISubscriber *) : HRESULT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+ClearDataSink(); HRESULT</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ISubscriberControl</th>
</tr>
</thead>
<tbody>
<tr>
<td>+Subscribe(sub : ISubscriber *) : HRESULT</td>
</tr>
<tr>
<td>+UnSubscribe(sub : ISubscriber *) : HRESULT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IServiceInstanceControl</th>
</tr>
</thead>
<tbody>
<tr>
<td>+SubscriberCount(count : int *) : HRESULT</td>
</tr>
<tr>
<td>+SubscriberName(num : int, name : char[10]) : HRESULT</td>
</tr>
<tr>
<td>+SubscriberChannel(num : int, channel : char[10]) : HRESULT</td>
</tr>
<tr>
<td>+SubscriberBindToPort(port : char[10], target : ISubscriberControl *) : HRESULT</td>
</tr>
<tr>
<td>+PublisherCount(count : int *) : HRESULT</td>
</tr>
<tr>
<td>+PublisherName(num : int, name : char[10]) : HRESULT</td>
</tr>
<tr>
<td>+PublisherChannel(num : int, channel : char[10]) : HRESULT</td>
</tr>
<tr>
<td>+PublisherBindToPort(port : char[10], target : ISubscriber *) : HRESULT</td>
</tr>
<tr>
<td>+PublisherUnBind(port : char[10]) : HRESULT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IHeartBeat</th>
</tr>
</thead>
<tbody>
<tr>
<td>+CheckAlive(); HRESULT</td>
</tr>
</tbody>
</table>

Figure 4.3 Interfaces

4.4.1 Previous interfaces

ISubscriber
The ISubscriber interface has not been changed.

IPublisher
There is one method added to the IPublisher interface: ClearDataSink(). This is the inverse of SetDataSink() and is used to clear the ISubscriber pointer that the publisher has. ClearDataSink() has no parameters.

IPublisherControl
The introduction of the Configuration Manager obsoletes the IPublisherControl interface, because the CM takes care of the adding, removing and replacing of Service Instances (and thus publishers).

ISubscriberControl
The ISubscriberControl interface has not been changed.

4.4.2 IHeartBeat

The IHeartBeat interface was added to provide the CM with a means of checking the SIs and brokers if they are still running. There is only one method in this interface: CheckAlive(), without parameters. If a call to CheckAlive() fails, the CM assumes that the SI/broker has crashed.

Called by: Configuration Manager
Implemented by: Service Instances and Brokers
Parameters:
*CheckAlive* (no parameters) method that is called by CM to check SI/broker. No functionality

### 4.4.3 IService

The IService interface only provides two methods: Start() and Stop() they are implemented by SIs to start and stop them.

**Called by:** Configuration Manager

**Implemented by:** Service Instances and Brokers

**Parameters:**
*Start* (no parameters) start the service/broker

*Stop* (no parameters) stop the service/broker

### 4.4.4 IServiceInstanceControl

The IServiceInstanceControl interface provides all methods needed by the CM to query a SI for the channels on which it wants to publish/subscribe. In addition, IServiceInstanceControl provides the methods for binding and unbinding the publishers and subscribers contained in the SI to brokers. The methods below marked with ¹ can be combined into one method (e.g. called QueryServiceInstance). This reduces the number of calls to one. However, for clarity these methods are listed separately.

**Called by:** Configuration Manager

**Implemented by:** Service Instances

**Parameters:**
*SubscriberCount* ¹ [out] int *count* returns the number of subscribers contained in this SI.

*SubscriberName* ¹ [in] int num the number of the subscriber

*SubscriberChannel* ¹ [in] int num the number of the subscriber

*SubscriberBindTo* [in] char[10] port the port that will be bound to the broker (this is the name as returned by SubscriberName)

*SubscriberUnBind* [in] char[10] port the port that will be bound to
PublisherCount

- **[out]** int *count*
  - Returns the number of publisher contained in this SI.

PublisherName

- **[in]** int *num*
  - The number of the publisher
  - $0 \leq \text{num} < \text{count}$

- **[out]** char[10] *name*
  - The name of the publisher identified by num

PublisherChannel

- **[in]** int *num*
  - The number of the publisher
  - $0 \leq \text{num} < \text{count}$

- **[out]** char[10] *channel*
  - The name of the channel identified by num

PublisherBindTo

- **[in]** char[10] *port*
  - The port that will be bound to the broker (this is the name as returned by PublisherName)

- **[in]** ISubscriber *target*
  - Pointer to the broker that the publisher must publish to

SubscriberUnBind

- **[in]** char[10] *port*
  - The port that will be bound to the broker

4.5 Components

This section describes the COM components used in the second phase. Note that the CM is no COM component. The CM is a COM client exposing no interfaces. Therefore, nothing is said about the CM, but only about brokers and SIs.

4.5.1 Broker

Figure 4.4 shows the broker component. Note that with respect to the broker component from the previous phase (see Figure 3.2) the IPublisherControl interface has disappeared. This is because the CM fulfills this role in this phase. For the rest nothing has changed.

![Broker component diagram](image-url)

Figure 4.4 Broker component

Phase 2: Replaceability and robustness
4.5.2 Generalized Service Instance

Because different SIs have different amounts of publishers and subscribers, this section describes a generalized SI component. Figure 4.5 shows a generalized SI component. Although only one ISubscriber and one IPublisher are drawn, there can be zero or more instances of both of them.

![Figure 4.5 Generalized SI component](image)

4.6 Sequence diagrams

This section describes and shows the interactions that take place in the system for some example scenarios. These are:

- Querying a service instance for its publishers and subscribers
- The initialization of the system
- The crashing of a broker and the recovery of the system
- The crashing of a service instance and the recovery of the system

4.6.1 Querying a service instance

The first scenario is the querying of a service instance. The CM queries each SI it instantiates for the publishers and subscribers it contains. Figure 4.6 (left) shows the UML sequence diagram for this scenario. At first, the CM creates the SI. It then asks the SI for the number of publishers it has. Then, it checks – for each publisher – its name and channel (topic). The name is used to uniquely identify the publisher. In the following sequence diagrams, the querying is abbreviated to the right part of the figure.
4.6.2 Initialization of the system

Figure 4.7 shows an example sequence diagram for initialization of the system. This particular system consists of two SIs that use one topic. The first SI publishes to this topic, whereas the second SI both publishes and subscribes to this topic.

1. The CM creates the first SI.
2. It queries the SI for its publishers and subscribers.
3. The CM checks if a broker should be started, this is not the case, because the first SI only contains a publisher to the topic.
4. The second SI is created and queried.
5. The CM checks again if a broker has to be started. In this case, the answer is yes and the broker is started.
6. The publisher in the first SI is bound to the broker that was just created.
7. Both the publisher and the subscriber in the second SI are bound to the broker.
8. Now the system is started.
4.6.3 Recovering a crashed broker

Imagine a system with two SIs using the same topic. One SI only publishes and the other only subscribes. At a moment in time the broker for that topic crashes – for some reason. Figure 4.8 shows the sequence diagram of the procedure that takes place. The light grey arrows in the figure indicate HeartBeat calls made by the CM.

1. The system is running normally.
2. The broker crashes.
3. This is detected by a failed heartbeat call made by the CM.
4. The publishers and subscribers are unbound from the crashed broker. Note that the SIs cannot distinguish if the broker has crashed, or is being replaced.
5. The CM creates a new broker for the topic.
6. Both the publishing and the subscribing SIs are bound to the new broker.
7. The system continues to function as before.
4.6.4 Recovering a crashed SI

Again, take the 2 SI system from the previous section. This time the subscribing SI crashes. Figure 4.9 shows the sequence diagram for this case.

1. The second SI crashes
2. This is detected by the heartbeat call.
3. The CM restarts a new SI (of the same as the crashed one).
4. The new SI is bound to the broker.
5. The system functioning continues.

Note that the broker is not informed about the crashed subscriber. This is detected by the broker itself when the next message arrives at the broker.

4.7 Prototype

4.7.1 Introduction

This section describes the prototype that implements the system described in the previous sections. The prototype is still based on text strings, but these are automatically generated
by two SIs. The first SI publishes the time on topic ‘A’, the second SI publishes the position of the mouse on topic ‘B’. There can be several publishers on either topic. There are several (decided by the user) output windows that display either the time or the mouse position.

4.7.2 Configuration Manager

The Configuration Manager performs all tasks as described in Section 4.3.1. Moreover, it present a menu to the user. From this menu, the user can start an arbitrary number of SIs, replace a SI or halt the system. The user can choose from the following SIs: ServiceTime, ServiceMouse and two types of output SIs (one for the time and one for the mouse position).

After a SI is started, the CM checks on which channels the SI wants to publish and on which topics it wants to subscribe. The CM checks whether publishers and subscribers exist on a topic. If this is the case the CM starts a broker and binds all publishers and subscribers for that topic to that broker. If a broker already exists for a topic to which a SI wants to subscribe or publish, the subscriber or publisher is bound to that broker.

4.7.3 Robustness

The robustness can be tested by using Window’s Task Manager. If a broker or SI is selected from the task list and terminated, the CM will detect a failure and report this on the screen. It then restarts the crashed broker or SI and restores all previous bindings to the broker or SI.

4.7.4 Redundancy

If more than one publishing SIs are started on a channel, redundancy is introduced. This makes the replacing of a publishing SI transparent, i.e. the windows displaying the time keep on getting updated. This can also be shown by the prototype.
5 Phase 3: Demonstrating the system

5.1 Introduction

The main goal of the third phase is to create an application demonstrating the use of the designed publish/subscribe system. For this application, a television simulator running on Windows 2000 was chosen. The next section will describe this simulator in more detail. The demo application should control the hardware simulation. It should be able to tune to other channels and change the zoom of the picture. In the running application, one SI must be replaced. This replacing should not be noticeable on the TV screen.

5.2 TV simulator

Inside Philips Research, a Visual Basic TV hardware simulator has been developed that simulates the actual hardware in an analog TV. The controlling of the hardware components is done by simulating an I²C connecting through a TCP/IP connecting. The remote control (RC) is also simulated by a TCP/IP connection. The prototype should connect to the simulators I²C server; the simulator connects to the RC servers implemented by the prototype.

Figure 5.1 shows the overview of the TV hardware. Not all hardware components will be used in the prototype, only the tuner (uv1316), high-end input processor (Hip), featurebox (Fbx), high-end output processor (Hop), the monitor, the Remote Control (RC) and the Broadcaster.

![Figure 5.1 TV system overview](image_url)

**Tuner**
The tuner is responsible for tuning to the signal. In the simulation, the signal is produced by the broadcaster component.

**Hip**
The Hip selects the signal from its inputs (either tuner or scarts) and decodes the signal.

**Featurebox**
The Featurebox is used for enhancing the image. It can for example move the picture on the screen, and can scale the picture. This is for example used to display a 4:3 picture full-screen on a 16:9 screen.
Hoop
The Hop is responsible for sending the signal to the screen, and doing final adaptations to the signal (e.g. contrast adjusting).

Remote Control
The remote control is used by the TV used to switch channels and zoom – in this system, in reality it can do much more.

Broadcaster
The broadcaster broadcasts the TV channels. This broadcaster is not in a TV, but only simulated. Normally, the channels are received from another medium (e.g. cable or satellite).

5.3 Prototype architecture

Figure 5.2 and Figure 5.3 show two options for the prototype architecture. The first option uses a broker for each communication link between two hardware-control SIs. The second architecture shows an architecture that uses a central broker for all communication links between hardware-control SIs. The first architecture uses more brokers (one per connection) whereas the second architecture uses only one. This means that in the second architecture all SIs must communicate through one broker. This is advantageous if all SIs must receive all messages. If this is not the case, it is easier to use the first architecture or to apply a filtering mechanism on the central broker (in the second architecture). The squares – in the software part – represent SIs, the circles represent brokers (with the corresponding channel name inside). The arrows indicate the data flow. For clarity, all control flows (from the CM to all SIs and brokers) have been omitted.

Figure 5.2 Broker per link
5.3.1 HW Control

The Hardware Control (HW Control) SI takes care of the TCP/IP connection to the hardware. Note that in both figures no broker is used for the communication between the SIs and the HW Control. This is because the calls from the SIs to the HW control are comparable to hardware calls (they are synchronous and blocking). The HW control is only drawn for clarity of how the communication with the hardware takes place.

5.3.2 Remote Control

The actual remote control is also simulated. The Remote Control (RC) SI is a TCP/IP server where the hardware simulation connects to. The RC SI receives the remote control codes from the hardware. It interprets the messages and sends them on to the control components through the Command (CMD) channel.
5.4 Design choices

5.4.1 No global knowledge

The operations such as tuning to another channel must not cause artifacts (e.g., snow) on the screen. This can be achieved by blanking the screen prior to performing any operation. Thus, there has to be a command to the Hop indicating that the screen has to be blanked, if an operation is going to take place. There are two ways to solve this:

1. Introduce a ordering on the SIs (like in Figure 5.2). This creates a chain, in which each SI has one or two neighbors. Then, if an operation has to take place, this is communicated to the right neighbor that can take appropriate action and communicate this further to its neighbor until it reaches the end of the chain. The end of the chain is the Hop that blanks the screen.

2. Have an SI that is aware of the total system and takes care of the sending of the Blank command to the Hop.

The second option introduces a SI with knowledge of the system. This is not desirable, because changes in the system (e.g., replacing an SI) cause the system architecture to change. Then, the SI with central knowledge has to be informed.

Using the first alternative distributes the global knowledge of the system to the individual SIs – the SIs do not have information about the total system, they are only aware of one (or two) neighbors. Therefore, the first option was chosen.

5.4.2 Architecture without central broker

The architecture that was chosen is the one without the central broker, thus the architecture with a broker per connection. The reason for this is that there is a specific order (see 5.7) in the messages that have to be sent (as chosen in the previous section). If a central broker is used, all SIs get the message, and not only the tuner’s neighbor.

5.4.3 Introducing an AndBroker

Another problem that occurs is that a SI wants to receive acknowledgements (e.g., the screen has been blanked) from its neighbors, in order to know when to perform the action it needs to take (e.g., tune to channel). But it does not want to know how many neighbors it has, because this introduces more coupling between the SIs. Therefore, a different kind of broker is integrated in the normal broker: an AndBroker. This broker performs a logical AND operation on the messages that are published through it: if all publishers have published the same message, it sends that message to its subscribers. Normally, a broker does not know the numbers of publishers, but because this AndBroker is integrated to the normal broker and the AndBroker is used between the SIs that control the hardware (Tuner, Hip etc.) only, the number of publishers (to the AndBroker) equals the number of subscribers on the normal broker.

In this prototype, there is no component with more than one neighbor. But a logical extension is adding audio to the system. This component would be (together with the HIP) a neighbor of the tuner. For this extension, the AndBroker is required.

5.4.4 Horizontal communication between components

The communication protocol that is used between the SIs is a subset of a protocol that is called horizontal communication (HorCom). All SIs are ordered (in the same order as the hardware). The ordering corresponds with the data-flow in the hardware. This is needed, because e.g. if the tuners want to tune, the screen has to be blanked first.

This protocol is based on three messages: DROP, RESTORE and OK. The initiating SI sends the DROP – indicating to make the signal invalid – message to its successor, this message flows through the system until it reaches the final SI. Each SI along the way can
take appropriate action (e.g. the last SI will blank the screen). The last SI returns an OK message. The OK message flows back to the initiating SI that then performs its action. After that, it sends the RESTORE message – indicating to make the signal valid again – to its successor, which takes appropriate action (if any). The RESTORE message also flows to the last SI, which unblankes the screen and returns an OK message (that flows back to the initiating SI).

See 5.7 for examples of the horizontal communication mechanism.

5.4.5 One horizontal communication process simultaneously

Only one horizontal communication process will be handled simultaneously. This solves a problem that occurs that an OK message is received that is intended for another request. This problem could occur when one DROP or RESTORE command ‘overtakes’ (i.e. arrives at the last SI before) a previous one. Then, the OK for the overtaking command will arrive before the OK for the overtaken command, possibly causing an error. See Section 5.7, Figure 5.5 for an example.

This decision was made for simplicity; there are other options to prevent the ‘overtaking’. For example, a SI could suspend the relaying of a DROP message until the OK for the RESTORE (that is in progress) has been received. Another option is to have a DROP message obsolete the current horizontal communication process.

5.4.6 Replacing a control component

There are two possibilities for replacing a control component. First, only allow replacing a control component when no HorCom sequence is in progress. This can be achieved by letting a SI not detach itself from its brokers, until the sequence has completed (at least for this component). After that, the SI can be replaced. But this introduces the problem that a message could be missed. If there is no component attached to the broker that sends the DROP/RESTORE messages, and a message arrives at that broker, there is no subscriber to deliver the message to. Having the broker remember the message until a subscriber is available can solve this. This, however, introduces the problem that if there should be two SI connected to that broker (as subscriber) and the broker does not know that, one SI could still miss that message (if that SI is being replaced at that moment). Therefore, it is required to notify the broker about the replacing of the SI.

But if this is necessary, it is also possible to replace a SI when a HorCom process is in progress. This needs an additional property of the system: transfer of state. This is discussed in Section 5.5. The replacing then takes place as follows: the CM notifies the brokers connected to the to-be-replaced SI that it is going to be replaced (using the Start/Stop methods). The brokers then suspend the sending of the message they received until they are notified that the replacing has been completed. After that, the SI is replaced and the brokers are notified.

5.5 Transfer of state

A desirable feature is the ability to replace components that have state. Therefore, a mechanism should be added that enables the transfer of the state of one component to another.

There are two replacing scenarios. The first scenario is that the component that is to replace another component is created before the to-be-replaced component is removed from the system. Logically, the second scenario is that the component that will replace the other one is created after the to-be-replaced component is removed from the system. It would be preferable to have a state-transfer mechanism that supports both options.

There are three state-transfers methods:
1. Using the CM
2. Using a separate state-transfer channel
3. Let the SIs take care for themselves
5.5.1 Using the CM

If the CM is used for the state-transfer, the CM obtains the state of the to-be-replaced SI and transfers this to the replacing SI.

Advantages
- Supports both replacing scenarios.
- Transparent to the system, CM can control the transfer.
- Can be used for robustness (saving the state now and then)

Disadvantages
- Requires the addition of GetState and SetState methods to the interfaces of SIs and brokers.
- Does not use publish/subscribe.
- Poses more responsibility on the CM.

5.5.2 Using a separate channel

The second method uses a separate channel (i.e. a broker) to transfer the state from one SI to another. This means that all SIs and brokers must implement both a subscriber and a publisher for this channel.

Advantages
- Uses publish/subscribe.
- Interfaces do not need to change

Disadvantages
- Each SI needs a publisher and a subscriber for the channel.
- Brokers do not have publishers and subscribers.
- The second replacing scenario is not possible.

5.5.3 Letting the SIs take care

The third option is to let the SIs take care of state-transfer themselves. This means that the system is not responsible for transferring the state, but that SIs implement some mechanism to retrieve the state from (or send its own state to) another SI.

Advantages
- No responsibility for the system.
- Can support both replacing scenarios.

Disadvantages
- Does not use publish/subscribe.

5.5.4 Issues

One issue is that a to-be-replaced SI or broker can receive messages that change its state, just after or while the state-transfer is taking place. Stopping the to-be-replaced SI/broker before initiating the state-transfer can solve this. The new SI/broker is started after the state-transfer has completed. However, this solution causes temporary unavailability of the affected SI/broker. This may not be a problem, because all connections have to be broken for the replacing anyway.

5.6 Interfaces

This section describes the interfaces used in this phase. With regard to the previously defined interfaces, nothing has changed. However, there are some new interfaces needed. These will be discussed in the following sections.
5.6.1 IHWControl

IHWControl is used by the hardware-control SIs to communicate with the hardware. It allows the SIs to perform write, read and writeread (simultaneous read and write) actions on the hardware.

**Called by:** all hardware control SIs  
**Implemented by:** HWControl

**Parameters:**  
*Write*  
- `int address`: address of the hardware component  
- `char * msgw`: command to send to the hardware  
- `int lenw`: length of the command  
- `int result`: result code from the hardware

*Read*  
- `int address`: address of the hardware component  
- `char * msgr`: command to send to the hardware  
- `int lenr`: length of the command (number of bytes to read)  
- `int result`: result code from the hardware

*ReadWrite*  
- `int address`: address of the hardware component  
- `char * msgw`: command to send to the hardware  
- `int lenw`: length of the command  
- `char * msgr`: command to send to the hardware  
- `int lenr`: length of the command (number of bytes to read)  
- `int result`: result code from the hardware

5.6.2 IHWUser

The IHWUser is used for passing a IHWControl pointer to components that need the access the hardware.

**Called by:** Configuration Manager  
**Implemented by:** all hardware control components

**Parameters:**  
- `SetHWControl [in] IHWControl * hwcontrol`: pointer to the HW Control SI

5.6.3 ISubscriberControl2

The ISubscriberControl2 interface is the same as the ISubscriberInterface. It is used solely to distinguish between the normal broker and the andbroker.

5.6.4 IStateTransfer

The IStateTransfer interface is used to transfer the state to and from an SI or broker.

**Called by:** Configuration Manager  
**Implemented by:** SIs, brokers

**Parameters:**  
5.7 Sequence diagrams

Figure 5.4 shows an example of the horizontal communication that takes place between the hardware control components. The TV user issues the command to switch to channel 1. The tuner (control components) receives that command and sends a DROP message to its neighbor. The DROP message flows through the system until it reaches the HOP. The HOP blanks the screen and sends the OK message back. Eventually, the OK is received by the tuner that tunes to the new channel. After that it sends the RESTORE message to its neighbor (the HIP). The HIP checks if the hardware has decoded the new signal. If the decoding is done, it sends the RESTORE message to the FeatureBox that in turn zooms to the appropriate format. After zooming, it sends the RESTORE message to the HOP that unblanks the screen. Then, the OK message flows back to the tuner.

Figure 5.4 Horizontal communication example

Figure 5.5 shows a scenario where a new tune request arrives before the current request has been handled. Note that this scenario can not occur, because of the design decision made in 5.4.5, this sequence diagram only shows the scenario that could occur if this decision had not been made.

The horizontal communication starts just as in Figure 5.4, but while the protocol is in progress, the user decides to tune again. This request arrives at the tuner, which initiates the DROP sequence. The DROP message arrives at the HOP before the RESTORE message. Therefore, the HOP first blanks the screen and sends the OK message. Then, the HOP unblanks the screen (because of the RESTORE message). The OK message for the blank message is received by the Tuner, which tunes to the new channel (the tuner cannot distinguish if this OK is for the DROP or for the RESTORE). In the mean time, the HOP has unblanked the screen. Thus, the screen is not blanked when the tuner tunes, this results in snow on the screen.
Figure 5.5 Failure example
6 Conclusions and future work

This chapter describes the conclusion that can be drawn from the defined system, as well as work that could extend the current system. All over this chapter, this publish/subscribe system is assumed, thus using a broker.

6.1 Replacing

The design of the system and the prototypes show that using a publish/subscribe based interaction style provides dynamic replacement of software components. But how is this compared to a function-call based system? Consider a system with \( n \) publishers and \( m \) subscribers (or in function-call based language: \( n \) data producers and \( m \) data consumers), see Figure 6.1.

![Figure 6.1 Publish/subscribe vs. Function-call](image)

Now, there are two items that can be replaced a publisher (producer) and a subscriber (consumer).

**Replacing a publisher**

In the case that a publisher must be replaced, the publish/subscribe system only has to remove (and add) one binding – the binding from that publisher to the broker. However, if the function-call system is used, \( m \) bindings have to be removed and added.

**Replacing a subscriber**

If a subscriber has to be replaced, the publish/subscribe system again requires only one removal (and addition) of a binding. The function-call system requires \( n \) bindings to be removed and added.

This replacing behavior is the result of the decoupling of the components (provided by the broker). If a publish/subscribe system is used that does not include this decoupling, just as much bindings have to be removed and added, to dynamically replace a publisher or subscriber, as a function-call based system.

6.2 Consequences

Using publish/subscribe as an interaction style introduces some problems. In the first place, as the third prototype showed, brokers need to implement more functionality. This time the AndBroker was introduced to solve this problem. For the central broker option, even more knowledge would be needed.

Secondly, if two components want to communicate one-to-one, a dedicated broker is needed. A single broker for several one-to-one connections could be used for this, but this requires the introduction of filtering at the broker.

Filtering is also needed for establishing two-way communication. Otherwise, two SIs have to use two topics for communicating, thus two times one-to-one communication.
However, filtering cannot solve the first problem. The AndBroker might be replaced by some kind of filtering, but the central broker problem cannot be solved this way. This is because then the broker must know which SIs are neighbors, i.e. should have knowledge of the system.

6.3 Fault-tolerance

The second phase showed that if stateless SIs are used, the fault-tolerance is good. If SIs with state are used, it is more difficult to provide a robust system. Stateless SI can be restarted when they fail, and continue functioning normally. But when a SI with state crashes, the state is lost. See the next section for more information about state and fault-tolerance.

6.4 Future work

Real-time systems
In order to test the performance of a publish/subscribe system in real-time system, a prototype must be built on such a system. This report does not conclude anything about performance. Also, little can be said about memory utilization. Because the current prototypes run on Windows 2000, a lot of memory is lost for supporting the Windows framework.

Broker-to-broker connections
In a distributed environment, it can be beneficial to have a system that has a broker per node (for an example see Figure 6.2), instead of one central broker. This reduces the number of remote calls to one per other node. However, this requires broker-to-broker connections, in which a broker subscribes to another broker. This is possible due to the ISubscriber interface that is used by the broker. One of the problems this raises is the bouncing of messages between brokers. This can occur if brokers are connected to each other as regular subscribers. Probably, a special kind of subscribe call will solve this problem. However, this has not been analyzed yet.

![Figure 6.2 Broker per node](image)

More general system
The system described in this report uses specific chosen publish/subscribe variations. For some applications, these variations might not be the most efficient ones. Therefore, it can be desirable to have a more general system. In which applications using the publish/subscribe system can choose from certain variations – that are best-suited for the specific interaction behavior needed by that application – offered by the publish/subscribe system.
State
The previous section stated the problems with fault-tolerance if SIs with state are used. There are a number of solutions for this. Saving the state every time it changes is one option. Furthermore, introducing redundancy can also help. But both of these require more resources and thus loss of performance. This is a consideration that must be made.

Configuration Manager
The system described in this report uses a single CM. Therefore, this is a single point of failure. Future work could analyze the use of several CMs or a distributed one.
References

[1] Smith, D.C., Vinoski, S.,
Object interconnections,
SIGS C++ Report, February 1997

[2] OMG Event Service Specification,
March 2001

Gryphon: an information flow based approach to message brokering,
IBM Research, 1998

[4] Fetzer, C.,
Fail-aware publish/subscribe communication in Erlang,

The many faces of publish/subscribe,

Type based publish/subscribe,

[7] Rajkumar, R., Gagliardi, M., Shu, L.,
The Real-Time Publisher/Subscriber Inter-Process Communication Model for Distributed Real-Time Systems: Design and Implementation,

[8] Carney, D. Lee, A., Yong Chen, Q.,
Publish/subscribe,
http://www.cs.brown.edu/courses/cs227/groups/pubsub

[9] Huang, Y., Garcia-Molina, H.,
Publish/subscribe in a mobile environment,
MobiDE 01, 2001

[10] Eugster, P.Th, Guerraoui, R., Sventek, J.,
DACss: abstractions for publish/subscribe interaction,
Proceedings of the European Conference on Object-Oriented Programming, 2000

Content-based publish/subscribe with structural reflection,

[12] Laar, F. van de, Chaudron, M.R.V.,
A dynamic upgrade mechanism based on publish/subscribe interaction,

[13] Rogerson, D.,
Inside COM,
Microsoft Press, ISBN: 1572313498,
February 1997
Appendix A: MIDL interfaces

//============================================
// ISubscriber
//============================================
| uuid(B33A.A2B-8015-43cd-A836-3782A740AF3B),
object |
interface ISubscriber : IUnknown |
  | import "unknown.idl"; |
  | HRESULT Update([in, size_is(length)] char * text,[in] short length); |
|
//============================================
// IPublisher
//============================================
| uuid(989E569C-F5A0-4d0c-8746-3737932DBEAD),
object |
interface IPublisher : IUnknown |
  | import "unknown.idl"; |
  | HRESULT Start(); |
  | HRESULT Stop(); |
  | HRESULT SetDataSink([in] ISubscriber * sink); |
  | HRESULT ClearDataSink(); |
|
//============================================
// ISubscriberControl
//============================================
| uuid(7D52FF73-6092-4b9a-A332-EFE0808476B),
object |
interface ISubscriberControl : IUnknown |
  | import "unknown.idl"; |
  | HRESULT Subscribe([in] ISubscriber * sub); |
  | HRESULT UnSubscribe([in] ISubscriber * sub); |
|
//============================================
// IService
//============================================
| uuid(2914EAFE-AEB7-4a87-ADE9-2E752FE6C6E),
object |
interface IService : IUnknown |
  | import "unknown.idl"; |
  | HRESULT Start(); |
  | HRESULT Stop(); |
//===============================================
// IHeartBeat
//===============================================

// uuid(269D3B83-42AA-49ce-9786-3A7F6E3FF63F),
// object

interface IHeartBeat: IUnknown
{
    HRESULT CheckAlive();
}

//===============================================
// IServiceInstanceControl
//===============================================

// uuid(D490AF5D-E68E-489e-95F0-4B50C3642FA1),
// object

interface IServiceInstanceControl: IService
{
    import "unknown.idl";

    HRESULT SubscriberCount([out] int * count);
    HRESULT SubscriberName([in] int num, [out] char name[10]);
    HRESULT SubscriberChannel([in] int num, [out] char channel[10]);
    HRESULT SubscriberBindTo([in] char port[10], [in] IServiceInstanceControl * target);
    HRESULT SubscriberUnBind([in] char port[10]);

    HRESULT PublisherCount([out] int * count);
    HRESULT PublisherName([in] int num, [out] char name[10]);
    HRESULT PublisherChannel([in] int num, [out] char channel[10]);
    HRESULT PublisherBindTo([in] char port[10], [in] IServiceInstanceControl * target);
    HRESULT PublisherUnBind([in] char port[10]);
}