Architectural design of modular ESB systems

Clermond de Hullu

February, 2013

Department of Computer Science
Chair Software Engineering

Supervisors
Dr. Luís Ferreira Pires
Dr. Ing. Christoph Bockisch

Academic Year
2012/2013

UNIVERSITY OF TWENTE.

Table of Contents

Chapter 1

Introduction

1.1 Motivation

Executives across all industries are demanding more value from their strategic business processes. An important aspect that helps improve the value of those business processes is the ability to flexibly and rapidly create and change them. To do this, it is important that different software applications that support these processes are able to communicate with each other. However, not all software applications are built with interoperability in mind. So, allowing all those different applications to interoperate can be both time-consuming and costly due to different transmission protocols and data exchange formats.

The last few years have seen some significant technology trends to solve this problem, such as Service-Oriented Architecture (SOA), Enterprise Application Integration (EAI), Business-to-Business (B2B), and Web services. These technologies have attempted to address the challenges of improving the results and increasing the value of integrated business processes [1].

SOA is the current state-of-the-art in IT application architecture, and addresses the challenges of increasing the value of integrated business processes. It does not prescribe any specific technology for its implementation and can be implemented using Web services, an Enterprise Service Bus (ESB), an application server platform or other middleware. The ESB draws the best traits from these and other technology trends [1].

An ESB is an integration infrastructure to facilitate SOA that can be used to connect and coordinate the interaction of a significant number of diverse applications. Commercially available ESBs provide support for many additional capabilities and can be used to solve a wide range of complex integration scenarios.

Our previous research [2] has shown that commercially available ESBs can be a suitable choice for large companies, but these solutions may be too expensive or too complex, and can introduce too much overhead for smaller companies.

In this research we used CompanyX as an example of a company for which the commercially available ESBs are not a suitable choice, and because of that is using a custom-built ESB solution called the Prodigy Communication Center (PCC) to communicate with third-party services.

The PCC needs to be improved because it is not flexible enough and some other weaknesses are the poor performance, the lack of log information and there is also a need for additional functionality.

1.2 Objectives

The main objectives of this Master project have been:

1. To identify the architectural principles for developing ESB systems with stringent flexibility and quality requirements.

2. To define an architecture for a custom ESB system and evaluate the architecture by means of a prototype.

1.3 Approach

The approach we took consists of the following steps:

- **Analysis of a typical ESB architecture.**
  We analyzed the architecture of the PCC and identified its limitations.

- **Requirements analysis.**
  We identified and interviewed the stakeholders of an ESB system and used the identified limitations to capture the requirements of our project.

- **Architectural design.**
  We designed a new architecture to fulfil the identified requirements.

- **Prototype implementation.**
  We implemented the functionality of the designed architecture in a prototype.

- **Case study.**
  We carried out a case study to test, demonstrate and evaluate the functionality of our prototype.

- **Prototype evaluation.**
  Finally, we evaluated the prototype by checking its fulfillment of the requirements.

1.4 Thesis Outline

The outline of the remainder of our thesis is as follows:

Chapter 2 analyzes the architecture of the PCC and identifies its limitations, which we have used to set up the requirements of our new system.

Chapter 3 defines and explains the requirements of our new system.

Chapter 4 presents our high-level design, and explains the architectural principles for developing ESB systems, as well as the components of our architecture that promote these principles.

Chapter 5 refines our high-level design by describing the high-level components in more detail.

Chapter 6 describes the implementation of our prototype.

Chapter 7 describes our case study and the implementation of the plug-ins that we have developed for our prototype to provide a solution for the integration scenario of our case study.

Chapter 8 demonstrates the management environment we designed to be able to configure our ESB using the implemented plug-ins to provide a solution for the integration scenario of our case study.

Chapter 9 evaluates our system by analyzing the results of our case study, and by verifying the fulfilment of the requirements.

Chapter 10 summarises our conclusions and gives directions for future work.

Chapter 2

PCC Architecture

This chapter analyzes the architecture of the PCC and identifies its limitations. We identify these limitations by analyzing the PCC source code, reading the available documentation and interviewing the PCC developers. We use the identified limitations to set up the requirements of our project.

This chapter describes the PCC and its components without describing a specific integration scenario because the way the PCC processes messages is the same for all integration scenarios.

This chapter is structured as follows:
Section 2.1 provides general background information about ESBs.
Section 2.2 provides a high-level description of the PCC.
Section 2.3 discusses the PCC components.
Section 2.4 explains the messaging infrastructure's limitations.
Section 2.5 describes the security limitations.
Section 2.6 describes the architectural limitations.

2.1 Enterprise Service Bus

The term ESB was first used by Sonic in 2002 [3] to refer to their SonixXQ product, which was an XML-enabled Message-Oriented Middleware (MOM) that was later re-branded as Sonic ESB. A few months later, Gartner called ESB a strategic investment and then soon many systems from integration servers to messaging products were re-branded as ESBs.

Existing literature [3] [4] [5] [6] proposes multiple definitions for an ESB, but in our research we define it as an integration infrastructure that is used to facilitate SOA.

An ESB combines service hosting, message transformation, protocol bridging and intelligent routing to connect and coordinate the interaction of a significant number of diverse applications [1].

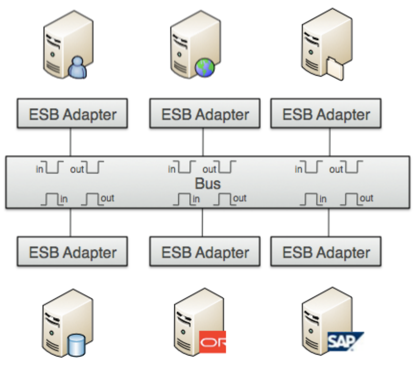


Figure 2.1: ESB Bus Architecture

This is done by overcoming differences in applications through mediation by allowing the applications to communicate over a bus-like infrastructure (as shown in Figure 2.1). ESB products support developers to build SOAs, but vary strongly in their operations and capabilities. More information about ESBs and their capabilities can be found in our previous research [2], which provides an ESB capability model that categorizes the ESB's capabilities.

2.2 PCC High-Level Description

The PCC is a custom ESB solution that was built to allow many different systems to communicate with third-party services in a standardized way, without these systems having to concern themselves with tasks such as message processing and message routing.

Figure 2.2 provides a context diagram depicting the high-level components that communicate with the PCC:

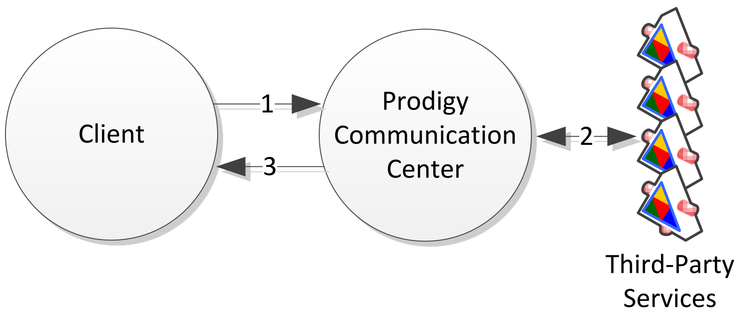


Figure 2.2: PCC Context Diagram

- **Client.** A system that uses the PCC to communicate with a third-party service.

- **Third-Party Service.** A service that provides functionality that is needed by a client.
- **Prodiy Communication Center.** A set of programs without a graphical user interface that facilitates the communication between clients and third-party services.

2.2.1 Message Flow

The message flow between the high-level components depicted in Figure 2.2 is as follows:

1. A client sends a message intended for a third party-service to the PCC. When the PCC receives the message it sends a synchronous response back to the client indicating if the PCC accepted the message, but this message does not contain the actual reply message from the third-party service.
2. Next, the PCC routes the received message to the correct third-party service and receives a synchronous reply message.
3. Finally, the PCC processes the reply message and routes it to the client that sent the original request message. For this to work, all clients have to host a Web service that accepts incoming SOAP messages from the PCC.

2.3 PCC Programs

The PCC has a decentralized architecture that consists of six different programs depicted in Figure 2.3 that have to be installed separately. Except for the Message Queue, they are all programs that host their own SOAP Web services to communicate with the other components.

Description of the PCC programs:

- **Incoming Frontend.** Hosts a SOAP Web service that all clients have to use to communicate with the PCC. This Web service provides a method that accepts messages containing the content that has to be sent to a third-party service.
- **Incoming Message Handler.** Receives messages from the Incoming Frontend and stores them in the inbox of the message queue.

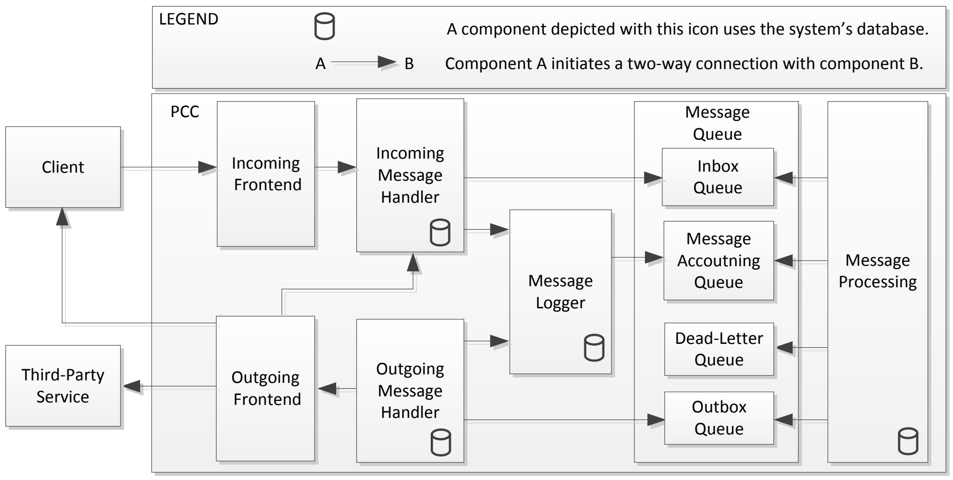


Figure 2.3: PCC Architecture

- **Message Queue.** A message queue implementation developed by Microsoft is used as the internal queue mechanism to prevent message loss when the PCC goes offline and it acts as a buffer that allows the PCC to accept messages faster than they can be processed. This is done by storing all incoming messages in the queue until they have been processed. The PCC uses a message queue, because it is faster than using the database.

- **Message Processing.** Reads messages from the inbox of the message queue, transforms the messages and finally adds routing information to the messages. If the processing of a message was successful it stores the message in the outbox of the message queue, if any errors occurred the message is stored in the dead-letter queue.

- **Outgoing Message Handler.** Reads the processed messages from the outbox of the message queue and sends them to the Outgoing Frontend.

- **Outgoing Frontend.** Sends the messages received from the Outgoing Message Handler to third-party services that communicate using HTTPS. It does this based on the routing information the Message Processing service added to the message. In addition, it also calls the SOAP Web service of the clients to deliver the actual reply messages from the third-party services.

- **Message Logger.** Stores log messages in the message accounting queue and then asynchronously reads those messages from the queue to store them in the database for persistence.

A more detailed explanation of the message flow can be found in Appendix A.2.

2.4 Messaging Infrastructure Limitations

This section discusses the limitations that we have identified related to the PCC messaging infrastructure.

L1: Not possible to host and communicate with additional endpoints.
It is not possible to host new endpoints that clients can use to communicate with the PCC without modifying its source code. This means that clients always have to call the Web service that is hosted by the Incoming Frontend (step 1 in Figure 2.2), and an even bigger restriction is that clients always have to host a SOAP Web service that accepts the reply messages from the PCC (step 3 in Figure 2.2). These are very strong restrictions that are imposed on all systems that need to be a client of the PCC.

For existing systems it is not always possible to adhere to these requirements, which prevents these systems from communicating with the PCC.

An example of this problem is when users store files on a FTP server, and the content of these files needs to be routed to a third-party service. Because a FTP server can not make SOAP requests and does not host a SOAP Web service to receive the reply messages it can never directly communicate with the PCC. So, to cope with this problem the PCC has to be able to host additional endpoints, which can use their own message format, MEP, communication protocol and endpoint address.

L2: Only support for a single communication protocol to communicate with third-party services.
The PCC is not capable of communicating with third-party services that use another communication protocol than HTTPS (step 2 in Figure 2.2). However, there are many third-party services that use another communication protocol. So, the PCC is strongly restricted since it can not communicate with these third-party services without having to modify its source code to implement the required communication capabilities.

L3: Customers have to open ports in their firewall.
When the PCC has successfully routed a message to a third-party service it needs to send the reply back to the client that sent the original message. To do this, it initiates a new connection with the client and sends the reply message (step 3 in Figure 2.2).

However, this causes many problems with the security policies of the customers running a client because they can only receive the reply message if they have opened inbound ports in their firewalls, which is often not allowed by their security policies.

L4: Clients are not notified if processing errors occur.
If the PCC accepted a message, but failed to process it, the client is never notified.

Instead of only storing the message in the dead-letter queue, the client must also be notified that an error occurred so that it can take necessary actions.

2.5 Security Limitations

This section discusses the security limitations that we have identified.

L5: General certificate for all reply connections.
A client must have the following three certificates installed for the encryption of the Web service calls and for the mutual authentication between the customers running a client and the PCC:

1. **PCC Certificate.** Allows a client to authenticate the PCC for the connection from a client to the Incoming Frontend.

2. **Client Certificate.** A unique client specific certificate, which allows the PCC to authenticate a specific client.

3. **Reply Certificate.** Allows the PCC to authenticate a client (but not a specific client) for the connection from the Outgoing Frontend to a client.

The reply certificate is a security vulnerability because it allows clients to decrypt reply messages that are addressed to another client. This is possible because the reply certificate is the same for all clients that use the PCC. This security limitation can be avoided by reusing the client certificate for the reply connection. This also simplifies the installation of a client because this reduces the number of required certificates to two.

L6: No authorization for different message types.

Using the previously discussed certificates all clients are authenticated and the communication with the Web services is encrypted, but there is no form of authorization. Once a client can establish a connection with the PCC it can send and receive all messages supported by the PCC.

2.6 Architectural Limitations

We have identified the following architectural limitations:

L7: Unexploited decentralized architecture.

The PCC has a decentralized architecture that consists of the six programs that are discussed in Section 2.3. However, CompanyX does not take advantage of the benefits of having a decentralized architecture, but does encounter the disadvantages.

For example, the six different programs need to be installed and configured separately using error-prone XML-based configuration files, but during the entire time that the PCC was being used not a single program was ever updated independently, all programs always ran on the same server and the programs are all very specific to the PCC, which prevents them from being reused.

So, because the architecture is very decentralized it requires more installation effort, it is harder to maintain and manage, and extending the PCC with a centralized location for management would provide a lot of overhead.

That is why a more centralized architecture would be more beneficial for CompanyX.

L8: Message routing rules can not be updated at runtime.

Message routing is done by the Message Processing program, and the criteria to identify a message type is hardcoded in its source code. The destinations of the different message types is stored in the database.

To modify the destination of an existing message type the destination address has to be changed directly in the database.

To add a new message type the following steps have to be taken: (1) the Message Processing program has to be stopped, (2) the code has to be modified, (3) changes have to be made in the database, and (4) the program has to be started again. This update disrupts the operation of all applications that are using the PCC.

L9: Message processing code can not be updated at runtime.
Message processing is also done by the Message Processing program. Modifying the message processing logic also requires its source code to be changed, which disrupts the operation of all applications that are using the PCC.

L10: Not possible to provide detailed and timely log information.
When using the PCC, looking for solutions to problems is very difficult and time consuming because the PCC does not provide detailed up-to-date log information on messages and their relations, and it is not possible to inspect which messages were sent by the specific external entities.

A cause of these problems is that the PCC logs all messages asynchronously. The advantage of this approach is that the message logging causes a negligible delay in the message processing, but the disadvantage is that there is not always up-to-date log information in the database. Because the PCC processes the log messages asynchronously with multiple threads, the order in which the log messages are received is lost, which prevents the PCC from logging the relations between the different log messages.

L11: No management application to manage the system.
A management application would reduce the installation effort, and make it easier for managers to maintain and manage the PCC.

Chapter 3

Requirements

This chapter describes the mandatory requirements of our custom ESB system, which is called the Prodigy Communication Center version 2 (PCC2).

The main purpose of the PCC2 is to reduce the time and cost of integrating third-party services by simplifying the development of applications that need to communicate with these third-party services.

This chapter is structured as follows:
Section 3.1 lists the stakeholders of the PCC2.
Section 3.2 describes our approach to gather the requirements.
Section 3.3 describes the mandatory requirements.

3.1 Stakeholders

The following stakeholders are relevant for the PCC2:

- **Developers** are the expert users that develop plugins for the PCC2, which allows the the PCC2 to communicate with additional clients and third-party services.
- **Service desk employees** install, configure, monitor and troubleshoot the PCC2.
- **Customers** indirectly communicate with the PCC2 via the client they use to communicate with a third-party service via the PCC2. They rely on the PCC2 to correctly process their messages and generate a reply for these messages.
- **Consultants** install and configure the clients for customers that communicate with the PCC2. So, they rely on the error messages and log information that the PCC2 provides to troubleshoot an installation.
- **Third-parties** provide services that rely on the PCC2 to retrieve or provide information.

By analyzing these stakeholders and their interests we identified that the PCC2 has the following three aspects that are of interest to the stakeholder:

1. **Management Application.** A program with a graphical user interface that can be used to manage the runtime engine.

2. **Runtime Engine.** The core of the ESB that communicates with third-party services, and performs task such as message routing and message processing.

3. **Development Tools.** A set of development tools that make it possible to extend the runtime engine's capabilities to communicate with additional third-party services.

Based on this observation we have identified the following three levels on which the stakeholders interact with the PCC2:

1. **Management Level.** Stakeholders that interact with the PCC2 on the Management level only communicate with the management application to retrieve and/or update information and settings.

2. **Operational Level.** Stakeholders on the Operational level only interact with the runtime engine by sending and receiving messages, but they are not allowed to use the management application.

3. **Development Level.** Stakeholders on the Development level do not directly communicate with PCC2, but they only make use the development tools.

For the purpose of this research it is enough to divide the stakeholders into the following three categories:

1. **Managers:** Service desk employees and consultants are managers and interact with the PCC2 on the Management level. Their main interest is to have a management application that provides all functionality they need to simply configure, monitor and troubleshoot the runtime engine.

2. **External Entities:** Customers and third-parties are external entities and interact with the PCC2 on the Operational level. Their main interest is to have the ability for their system to communicate with the runtime engine, so that it can correctly processes their messages to retrieve or provide information.

3. **Developers:** Developers interact with the PCC2 on the Development level. Their main interest is that the runtime engine provides tools that allow them to extend the runtime engine's ability to communicate with additional external entities with as little effort as possible.

Figure 3.1 depicts the interaction between the identified stakeholder categories and the PCC2.

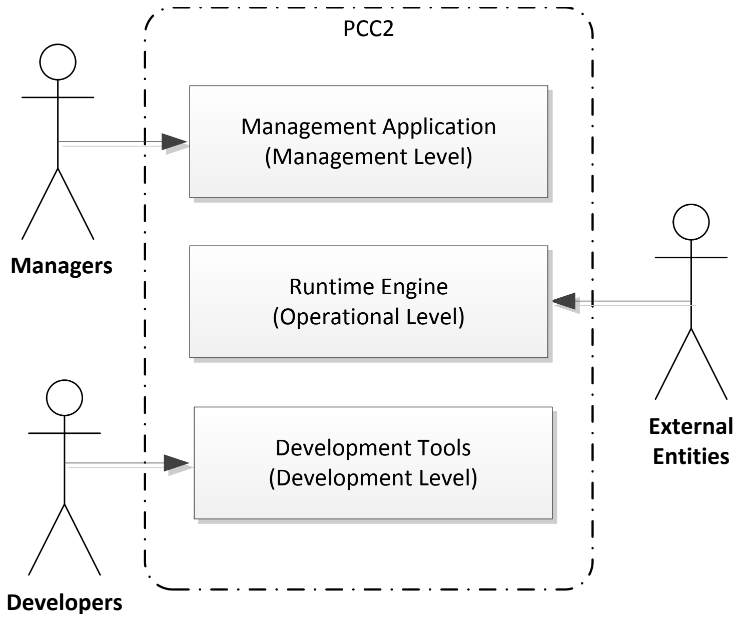

PCC2
Management Application (Management Level)
Runtime Engine (Operational Level)
Development Tools (Development Level)
Managers
Developers
External Entities

Figure 3.1: Stakeholder Categories

3.2 Approach

To specify the requirements of our project we interviewed the people that were identified by our project supervisor. We started by interviewing the PCC developers because they have the most knowledge of the usage, the features and limitations of the PCC. Next, we interviewed consultants and service desk employees because they use the PCC daily.

The first round of interviews were informal, and the main goal was to ask the stakeholders about their experience with the PCC, and if they had any thoughts or ideas that we had to consider when designing the PCC2. This allowed us to identify the main interests of the different stakeholders.

Based on these interviews we noticed that the consultants and service desk employees had the most interest in adding additional features, such as a graphical user interface to manage and monitor the PCC.

The developers were more interested in improving the PCC architecture to increase its flexibility of integrating new third-party services without disrupting the already running applications.

Based on the first round of interviews and the analysis of the PCC, we decided in consultation with our project supervisor that developing a new system with an improved architecture was more beneficial than adding additional features to the PCC.

Because of this decision we identified the developers as the main stakeholders of our project. To specify the requirements we performed a second round of interviews during which we re-interviewed the developers. This time we discussed the identified limitations of the PCC and determined the priorities for addressing them in the PCC2.

Based on the results of the interviews we compiled a lists of mandatory and optional requirements. The mandatory requirements address all limitations, except for L6 (no authorisation). This limitation is addressed by an optional requirement(R12), because this has to make use of an external licensing system that is still under development.

Verification of the fulfilment of the identified mandatory requirements took place by using the requirements as the evaluation criteria for the evaluation of our prototype.

3.3 Mandatory Requirements

This section describes the mandatory requirements of the PCC2. For readability we refer to the PCC2 as "the system" in the requirement descriptions.

Appendix A.1 describes the optional requirements that we have taken into account while designing the PPC2.

All requirements are specified in the following parts:

1. A short description which is used to refer to any requirement. The short description is prefixed with the letter "R" or "OR", which indicates if it is a mandatory or optional requirement.

2. The most important stakeholders of the requirement

3. The limitations that the requirements addresses.

4. Detailed description with additional information about the requirement.

Because the PCC2 must be capable of performing all the tasks the PCC performs, this chapter only describe the requirements that address the PCC limitations. The requirements are grouped according to the three identified levels on which the stakeholders communicate with the PCC, and are discussed in the following sections.

3.3.1 Development Level Requirements

The mandatory requirements for the interaction with the PCC2 on the Development level are:

R1: The system has to provide a mechanism to host and communicate with additional endpoints without disrupting the already running applications.

◇ *Stakeholders:* Developers, External Entities.

▹ *Addresses: L1: Not possible to host and communicate with additional endpoints.*

▹ *Addresses: L2: Only support for a single communication protocol to communicate with third-party services.*

This mechanism has to allow developers to develop and deploy reusable components that can be used to host or communicate with additional endpoints without disrupting the connections with the endpoints that are already operational.

Because endpoints can have different characteristics, this mechanism must be able to support a different MEP, communication protocol and message format for each endpoint that has to be interacted with.

R2: The system has to provide a flexible mechanism to perform message processing.

◇ *Stakeholders:* Developers.

▹ *Addresses: L9: Message processing code can not be updated at runtime.*

This mechanism has to allow developers to develop and deploy reusable components that can be used to implement custom business logic or perform message processing. The message processing options have to include:

- Performing XSD-Schema validations.
- Calling Web services or databases to find additional information to be added to the message.
- Applying XSLT transformations to transform the messages.

The developers should be able to update the message processing behaviour at runtime without modifying the system's source code or disrupting already running applications.

3.3.2 Operational Level Requirements

The mandatory requirements for the interaction with the PCC2 on the Operational level are:

R3: The system has to provide a mechanism to support asynchronous communication without external entities having to open inbound ports in their firewalls.
◇ *Stakeholders:* External Entities.
▹ *Addresses: L3: Customers have to open ports in their firewall.*
▹ *Addresses: L4: Clients are not notified if processing errors occur.*
▹ *Addresses: L5: General certificate for all reply connections.*
The system must provide a mechanism that allows external entities to communicate with the system without having to open any inbound ports in their firewalls, even if the system can not synchronously send the response. This mechanism must not use the reply certificate discussed in Section 2.5 to reduce potential security risks, and senders of messages must be able to verify that their messages have been processed successfully.

3.3.3 Management Level Requirements

The mandatory requirements for the interaction with the PCC2 on the Management level are:

R4: The system has to provide a flexible mechanism to perform message routing.
◇ *Stakeholders:* Managers.
▹ *Addresses: L8: Message routing rules can not be updated at runtime.*
The system has to be able to route messages using the following two approaches:

- **Itinerary-based routing.** The system must allow managers to use the management application to specify message itineraries that the system can use to route messages to their recipients.

- **Content-based routing.** The system has to be able to route messages based on the content of the messages.

The routing rules of both mechanisms have to be editable at runtime without modifying the system's source code, and without disrupting running applications.

R5: The system must only accept incoming messages from predefined external entities.
◇ *Stakeholders:* Managers.
The system must provide a mechanism that allows managers to specify the external entities that are allowed to communicate with the system, and only messages from those external entities must be accepted.

R6: The system has to be able to temporarily reject incoming messages.
◇ *Stakeholders:* Managers.
A manager has to be able to use the management application to indicate that the system must temporarily reject all messages.

R7: The system has to provide a configurable retry mechanism.
◇ *Stakeholders:* Managers.
The messaging infrastructure has to provide a configurable retry mechanism which allows managers to use the management application to specify:

- The amount of times a message has to be re-sent to an endpoint.
- The time the system has to wait after a failed sent attempt.
- The maximum time the system can take to process a message before it is discarded.

R8: The system has to be able to display detailed log information.
◇ *Stakeholders:* Managers.
▹ *Addresses: L10: Not possible to provide detailed and timely log information.*
The system has to log the complete flow of messages through the system and allow managers to view this flow by using the management application.
This flow should include:

- The name of the external entity that sent a message.
- At what time a message was sent to a recipient and the reply that was received for the message.
- Which errors, if any, occurred during the processing of a message.
- The final reply message that was generated by the system.

This makes it possible to view the flow of messages through the system and allow managers to see what exactly has happened in case something went wrong. This information also makes it possible to provide statistics and usage information.

Chapter 4

High-Level Design

This chapter presents our high-level design of the PCC2, and it describes the architectural principles for developing ESB systems together with the components of the PCC2 that promote these principles.

This chapter is structured as follows:

Section 4.1 discusses the two most important architectural principles for developing ESB systems.

Section 4.2 provides a high-level overview of our architecture.

Section 4.3 describes the high-level components in more detail.

Section 4.4 describes the publish-subscribe engine.

Section 4.5 discusses the adapter framework.

Section 4.6 explains the message processing capabilities of the PCC2.

Section 4.7 deals with security.

4.1 Architectural Principles

We have identified the architectural principles for developing ESB systems by analyzing the architectures and documentation of the existing ESB systems that we have discussed in our previous research [7] [8] [9] [10] [11] [12] [13] [14].

The documentation of Neudistic's Neuron ESB [15] [16] [17] and Microsoft's BizTalk Server [18] [19] [20] [21], which are the two state-of-the-art commercially available .NET-based ESB solutions, provided the most detailed and extensive information about ESB architectures.

Based on our analysis we concluded that providing mediation and a flexible messaging infrastructure are the two most important architectural principles for developing a custom ESB system. These two principles are discussed in the next two sections.

Our high-level design covers the remaining architectural principles together with the components of the PCC2 that promote these principles.

4.1.1 Flexible Messaging Infrastructure

Enterprise developers, traditionally, write applications that include both business logic and communication logic. In the past, developers of communication logic made assumptions about from which systems information came and also to which systems information should be sent. The problem with that model is that it requires the developer to have an enterprise-level view of how systems interconnect, and that understanding is then embedded into the code. If the connections change, the application has to change as well.

ESBs provide a publish-subscribe engine to take that burden off of the developer and make it a concern of the business analysts and the IT department. When a publish-subscribe engine is used, the routing of messages is determined by a centralized component, which allows the routing to be changed at any time, without requiring any changes to application software.

Publish-subscribe is a messaging pattern in which senders of messages, called publishers, do not program the messages to be sent directly to specific receivers, called subscribers. Instead, published messages are put into classes without knowing if there are any or how many subscribers are interested in these messages, and subscribers only receive a subset of the total number of messages which are published, the subset in which they are interested.

Filtering is the process of selecting messages for reception and processing. There are two common forms of filtering namely topic-based and content-based. In a topic-based system, messages are published to "topics" or named logical channels. Subscribers in a topic-based system receive all messages published to those particular topics to which they subscribe, and all subscribers to a specific topic receive the same messages.

In a content-based system, messages are only delivered to a subscriber if the attributes or content of those messages match the constraints defined by the subscriber.

4.1.2 Mediation

In order to make flexible messaging possible, the ESB has to mediate the communication between all applications that need to communicate with each other.

There are multiple forms of mediation, which together help overcome differences in applications, and allow the different applications to communicate through the ESB. An ESB has to bridge differences in protocols, message formats, security models and communication semantics. It is much simpler to configure an ESB for mediation than it is to make changes to the programs themselves.

Below we discuss the four different types of mediation that an ESB has to provide to allow different applications to communicate through the ESB.

1. **Transformation** converts messages into a format understood by the receiver, and is the key tool for supporting message versioning. Transformation can also be used to reconcile differences in messages to or from parties who may have differing requirements about format and content.

2. **Protocol mediation** overcomes differences in a communication protocol to for example connect a program that sends via queuing to a service that receives via HTTP. Differences in communication semantics may also need to be mediated, as in connecting a program that expects a request-reply pattern of communication with a program that performs one-way communication.

3. **Security mediation** handles differences in security models, as in the case of an enterprise application that uses Windows integrated security conversing with a service that uses X.509 certificates.

4. **Time mediation** handles differences in time. A sender may be transmitting messages when some receivers are not currently available. Time mediation can store messages until a receiver is available to receive them, even though the sender and the receiver might be using non-durable protocols such as HTTP. A characteristic component of an ESB to mediate differences in time is a buffering component that usually is implemented as a message queue that allows the ESB to cope with differing handling speeds.

4.2 High-level Components

An ESB has to promote configuration changes rather than program changes because configuration changes are simpler, safer, and can be applied without disrupting the running applications. To do this we designed the PCC2 with the two components depicted in the context diagram in Figure 4.1. This context diagram depicts the PCC2 high-level components and the stakeholders that communicate with them.

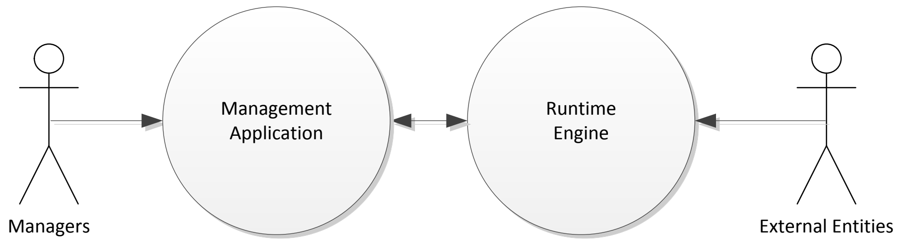


Figure 4.1: PCC2 Context Diagram

These components are:

- **Runtime Engine.** A program without a graphical user interface that executes the tasks received from the management application and provides the types of mediation discussed in Section 4.1.2 to allow different applications to communicate over a bus-like infrastructure.

  The runtime engine is discussed in more detail in Section 4.3.

- **Management Application.** A program with a graphical user interface that managers can use to communicate with a runtime engine to make configuration changes, monitor, and control the runtime engine with a single application.

  The management application's interface and functionality is discussed in Chapter 8.

4.3 Runtime Engine

One of the challenges we encountered when designing the runtime engine was to allow it to flexibly communicate with many different endpoints, while still keeping it doable to design and implement a prototype in the time frame of this project. To do this we designed a custom adapter and workflow framework. These frameworks allow the developers to extend the PCC2 to interact with new endpoints without disrupting the communication with already operational endpoints using techniques with which they are already familiar. This reduces the learning curve of using the PCC2 and it prevents us from having to implement many different mechanisms to communicate with endpoints the PCC2 might need to communicate with in the future.

Figure 4.2 depicts the runtime engine's high-level components together with examples of adapters and workflows that can be developed for the PCC2. The runtime engine consists of the components depicted within the dashed line, which are

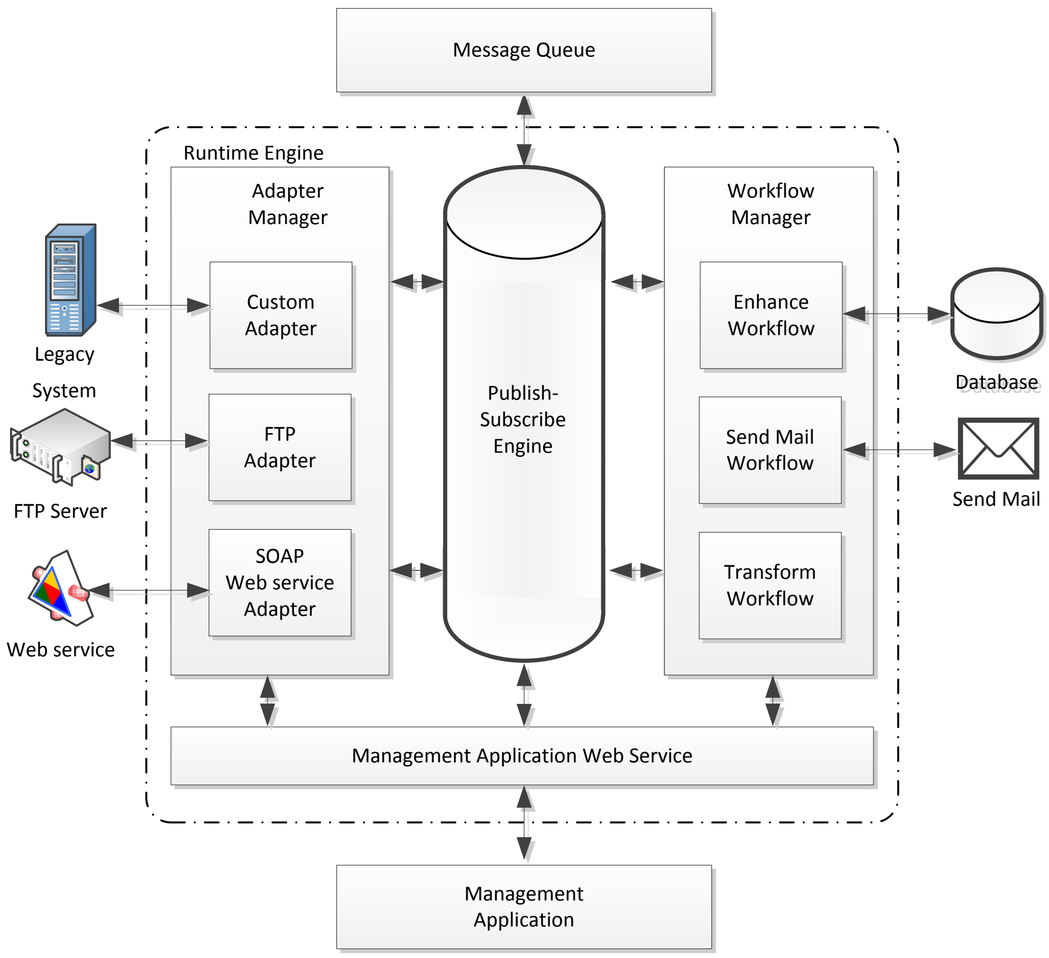


Figure 4.2: High-Level Architecture

implemented as modular, replaceable, and extensible .NET classes. The runtime engine communicates with a database, a message queue and the management application.

Below we give a short description of the runtime engine's high-level components:

- **Publish-subscribe engine.** Provides a flexible messaging infrastructure that can mediate differences in time and communication sematics, and provides a central location to perform message routing which is fully configurable using the management application.

  The publish-subscribe engine is discussed in Section 4.4.

- **Message Queue.** Allows the publish-subscribe engine to store messages in a message queue to cope with differing handling speeds and prevents message loss when the runtime engine is unexpectedly restarted.

- **Adapter Manager.** Allows developers to develop and deploy adapters that allow the runtime engine to host new endpoints or interacting with new endpoints without disrupting the communication with already operational endpoints, and provides security and communication protocol mediation.

  The adapter framework is discussed in Section 4.5.

- **Workflow Manager.** Provides the framework to deploy and execute workflows that can be used to perform message processing or implement flexible runtime updatable business processes without disrupting the already running adapters and workflows.

  The workflow framework is discussed in Section 4.6.

- **Management Application Web Service.** A SOAP Web service that is hosted by the runtime engine and provides all the functionality that is needed by the management application to manage the runtime engine remotely.

The runtime engine isolates and decouples communicating parties (adapters and workflows), allowing them to effectively exchange messages by simply publishing messages onto the bus, without regard to the type or number of consumers; similarly, they may subscribe to specific topics, without regard to the source of the messages. This frees the developers from spending time on messaging business logic and allows them to concentrate on the specific business logic associated with manipulating the message data.

Design Alternative

An alternative to implementing the runtime engine's different components as .NET classes is to implement them as different programs that communicate via SOAP Web services. This provides a more decentralized architecture which increases scalability. However, based on the usage of the PCC (L7: Unexploited decentralized architecture) we have determined that for our project having a more centralized architecture is more beneficial.

The additional overhead of a decentralized architecture is required for large enterprise systems that need their components to be able to run on different servers for scalability, but benefits do not outweigh the drawbacks for our project.

The centralized approach has the following advantages compared to the more decentralized architecture:

- Performance improvement because the communication between components that run in the same process have less overhead than communicating via Web services.
- Less installation effort because there is only a single program that has to be installed and configured.
- Quicker development because the communication between Web services takes more development and test effort than communication between components in the same process.
- Easier to maintain and less overhead to provide a centralized management application that can monitor and control the runtime engine to promote configuration changes rather than program changes.

4.4 Publish-Subscribe Engine

We chose to design a topic-based publish-subscribe engine because this makes it possible to use the management application to configure the subscribers for each topic based on the name of the topic. A topic can be something as simple as for example "Orders". Publishers can label each message with this name and publish the messages to that topic. This makes the configuration of the subscribers straightforward, and subscribers can still choose to ignore specific messages if the content does not meet their requirements.

Topics Configuration

Each topic has its own configuration that the publish-subscribe engine uses to determine how to publish messages to a specific topic. The default behaviour is that the publish-subscribe engine publishes the same message to all subscribers of a topic in the order in which a manager specified the subscribers.

To allow the runtime engine to prevent message loss, and communicate with additional endpoints that use their own MEP, we made the publish-subscribe engine more flexible by providing the following extensions:

- **Request-Response Messaging.** Provides support for synchronous request-response messaging in addition to the default asynchronous publish-subscribe messaging.

- **Chaning.** Provides functionality that alters the default publishing behaviour by allowing publishers to manipulate messages while they are being published, which makes it possible to use the output of a subscriber as the input of the next subscriber.

- **Message Box.** Provides support for asynchronous messaging without external entities having to open inbound ports in their firewalls.

- **Queuing.** Provides functionality that allows the runtime engine to cope with differing handling speeds and prevent message loss when the runtime engine is unexpectedly restarted.

The publish-subscribe engine and its specific features is covered in more detail in Section 5.1.

Internal Message Format

We designed an internal message format to exchange data between all publishers and subscribers. The publish-subscribe engine only communicates via this internal message format, which makes it possible to communicate with all publishers and subscribers using a standardized way.

An internal message is serializable to XML, and contains both data i.e., the information that some other system, resource or person may be interested in, as well as metadata.

An important benefit of adopting XML as the means for exchanging data between publishers and subscribers is the extensibility of XML, which allows portions of a message to be modified or enhanced without affecting other portions of the message.

In addition, techniques such as XSTL and XPath expressions can provide a certain degree of loose coupling between the publishers and subscribers.

4.5 Adapter Framework

It is not known at design-time which systems should communicate with an ESB, so there may be systems that have to communicate with the ESB, but cannot yet be supported.

To cope with this problem the PCC2 provides a flexible adapter framework that allows developers to develop and deploy custom adapters that can be used to host new endpoints or to interact with new endpoints. The adapter framework provides this functionality without requiring the PCC2 source code to be changed and without disrupting the connections with the endpoints that are already operational.

Adapters can be implemented by simply extending a base class that provides functionality to communicate with the publish-subscribe engine. An adapter encapsulates the logic to communicate natively with a target endpoint and also provides the functionality to communicate with the publish-subscribe engine, so with the adapter in place the publish-subscribe engine and the target endpoint can interact. An adapter is capable of sending and receiving messages, resides on the server, and runs as part of the runtime engine's process.

Before an adapter can be used, it must first be registered. This allows the runtime engine to check its configuration by verifying that all registered adapters are still available, and this makes it possible to enforce security policies on the registered adapters.

Once registered, multiple instances of an adapter can be created using the management application. Adapters can declare properties, and for each adapter instance it is possible to specify different values for the declared properties. This makes it possible to create multiple instances of a registered adapter that behave differently.

In addition, the PCC2 provides a Service Host Factory that adapter developers can use to host a SOAP Web service that external entities can use to communicate with an adapter with a single line of code.

The runtime engine contains an Adapter Manager (see Figure 4.3) that provides all the functionality that the publish-subscribe engine needs to interact with adapters. The Adapter Manager dynamically loads all the developed adapters from a directory on the file system and instantiates all adapter instances with the properties that a manager specified using the management application.

4.6 Workflow Framework

The PCC2 provides a framework that allows developers to implement visual workflows that can be deployed and executed by the runtime engine.

These workflows receive an internal message as parameter, and can be used to perform many tasks such as service composition, message processing, executing custom business logic, sending emails, or enhancing messages with routing information.

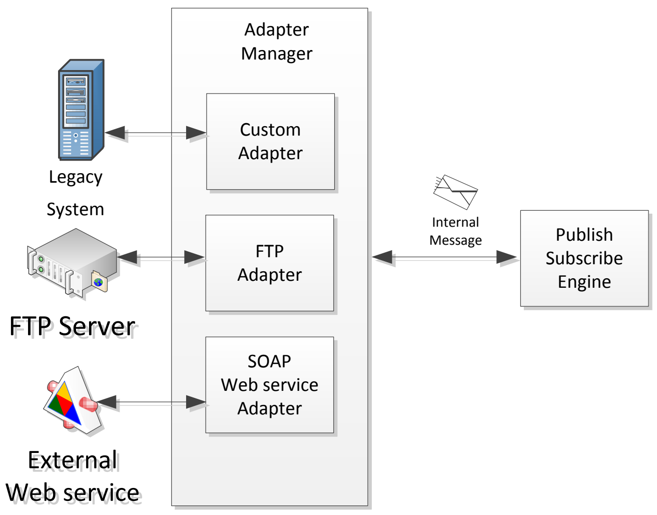


Figure 4.3: Adapters Manager

They can be updated at runtime without disrupting the already running adapters and workflows, which makes them a flexible solution that provides a wide range of capabilities.

Because the adapters create internal messages, the workflows only have to deal with the internal message format, and they do not need to be aware of endpoint specific-properties, such as the communication protocol of an endpoint.

However, they may need to transform the content of messages because the structure of the received message is not in the expected format, additional information is needed, or the message must be converted, e.g., from XML to a non-standard format.

The runtime engine contains a Workflow Manager (see Figure 4.4) that provides all the functionality that the publish-subscribe engine needs to interact the with developed workflows. The Workflow Manager can communicate with the developed workflows that are loaded from a directory on the file system, and can communicate with the publish-subscribe engine using the internal message format.

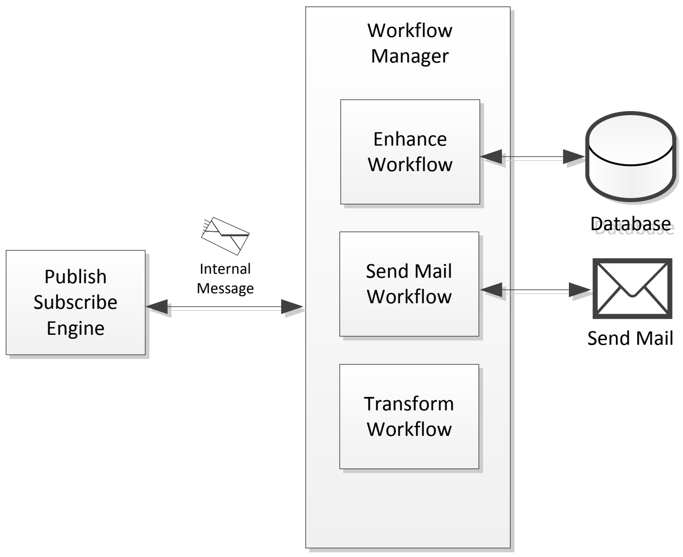


Figure 4.4: Workflow Manager

Just like adapters, workflows have to be registered first, and once registered, a workflow can be used as a subscriber of a topic.

4.7 Security

To allow the runtime engine to provide security and usage information, all incoming messages are linked to an external entity that represents the sender of a the message. External entities have to be specified by a manager using the management application and can have different permissions.

Because the runtime engine uses adapters to encapsulate endpoint specific logic to communicate with multiple different endpoints in a standardized way, the adapters are also responsible to encapsulate the different security mechanisms of endpoints.

So, adapters have to provide transport level security between the runtime engine and an endpoint, and the publish-subscribe engine performs the authentication and authorisation of external entities to prevent adapter developers from having to include this logic in the adapter's implementation.

The publish-subscribe engine does this based on information adapters have to specify in the metadata of internal messages that are published to the publish-subscribe engine. If the publish-subscribe engine receives an internal message from an adapter it tries to find an external entity based on the provided metadata. If no external entity is not found, the message is discarded.

Chapter 5

Architectural Design

This chapter refines the high-level design by describing the high-level components in more detail. To develop the system in a systematic way we have used Domain-driven design (DDD), which is an approach to develop software for complex needs by connecting the implementation to an evolving domain model.

This chapter discusses the domain model step-by-step and provides an overview of the complete domain model. This domain model is also used to create the data model by using an object-relational mapping (ORM), which is a mechanism that makes it possible to address, access and manipulate objects without having to consider how those objects relate to their data sources.

This chapter is structured as follows:
Section 5.1 addresses the part of the domain model that is used for the topics and their subscribers.
Section 5.2 discusses the part of the domain model that is used for message logging.
Section 5.3 discusses the retry mechanism.
Section 5.4 describes the request-response messaging extension of the publish-subscribe.
Section 5.5 describes the chaining extension of the publish-subscribe engine.
Section 5.6 describes the asynchronous messaging extension of the publish-subscribe engine.
Section 5.7 describes the queuing extension of the publish-subscribe engine.
Section 5.8 deals with security.
Section 5.9 describes the flow of a message through the runtine engine.

5.1 Topic Domain Model

Figure 5.1 depicts the part of the domain model that the publish-subscribe engine uses for the topics and their subscribers. A short description of the classes is given below:

- **Topic.** Describes a topic and its configuration.

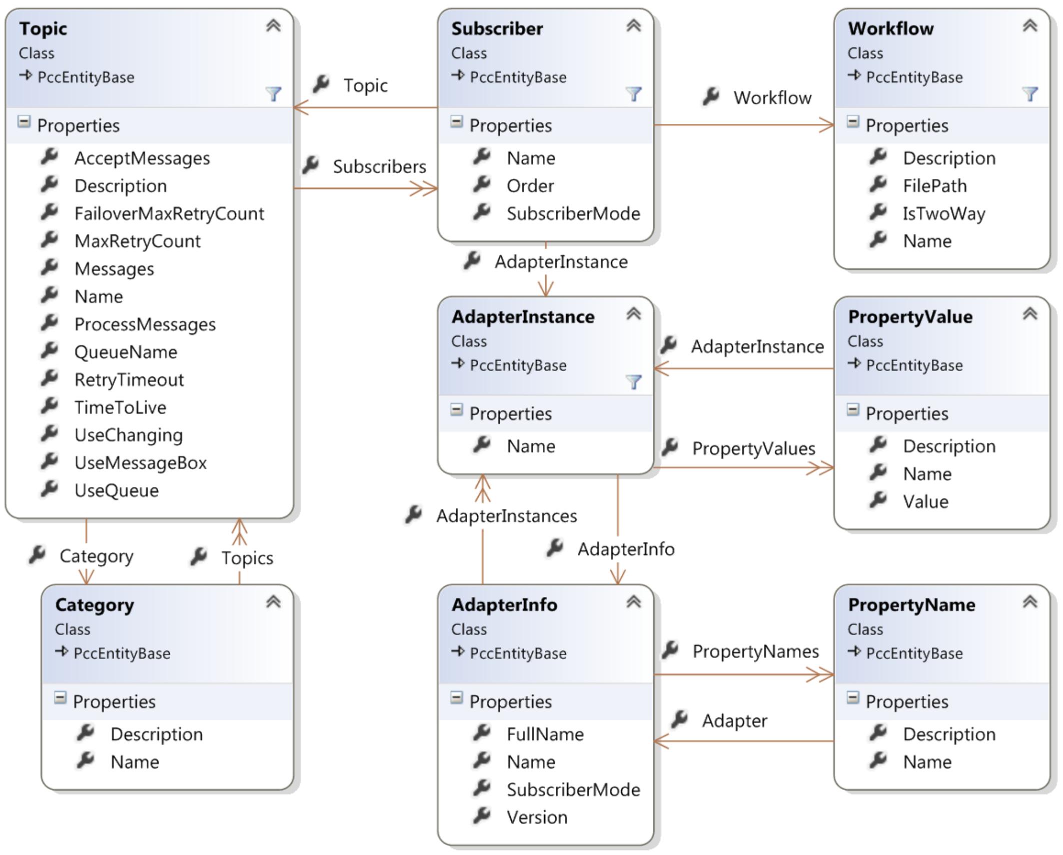


Figure 5.1: Topic Model

- **Category.** Topics belong to a category, which makes it possible to group topics in the management application.
- **Workflow.** Describes a workflow implementation stored in the file system.
- **Adapter Info.** Describes an adapter implementation stored in the file system.
- **Adapter Instance.** Describes an instance of an adapter.
- **Property Name.** Describes the name and description of a property declared by an adapter.
- **Property Value.** Describes the name, description and value of a property configured for an adapter instance.
- **Subscriber.** Represents a workflow or adapter instance that can be a subscriber of a topic.

Topic Configuration

The publish-subscribe engine publishes messages depending on the configuration of the topics to which the messages are published. Managers can use the management application to create, view, update and delete the topics used by the publish-subscribe engine. Each topic can have a different configuration which consists of the subscribers to the topic, and the following settings:

- **Name.** The unique name of a topic.
- **Description.** The description of the topic.
- **Accept Messages.** Specifies if the topic is currently accepting messages. If this property is set to false all incoming messages for this topic are rejected.
- **Time To Live.** The maximum time in seconds the publish-subscribe engine may take to process messages before they become unusable and have to be discarded.

The publish-subscribe engine provides a retry mechanism that can be configured with the following settings:

- **Max Retry Count.** The number of times the publish-subscribe engine retries to send a message to a subscriber.
- **Retry Timeout.** The time in seconds the publish-subscribe engine waits before the message is re-sent after a failed attempt.
- **Failover Max Retry Count.** The number of times the publish-subscribe engine retries to send a message to a subscriber using the fail over mode of an adapter.

The extensions of the publish-subscribe engine can be enabled and configured with the following settings:

- **Use Message Box.** Specifies if the message box mechanism, which provides asynchronous messaging capabilities, must be used.
- **Use Chaining.** Specifies if the chaining mechanism, which determines how messages are published to the configured subscribers, must be used.

- **Use Queues.** Specifies if the queuing mechanism, which allows messages to be buffered in a queue, must be used. If this setting is enabled the following properties have to be specified as well.
  - **Queue Name.** The name of the queue that has to be used.
  - **Process Messages.** Specifies if the messages stored in the queue have to be processed. If this setting is set to false new incoming messages are still accepted and stored in the queue, but they are not processed until this setting is enabled.

5.2 Message Logging

To allow managers to troubleshoot problems, and to provide usage information all sent and received messages are logged to a database. Figure 5.2 depicts a simplified view of the discussed internal message format, which we called a Communication Center Message.

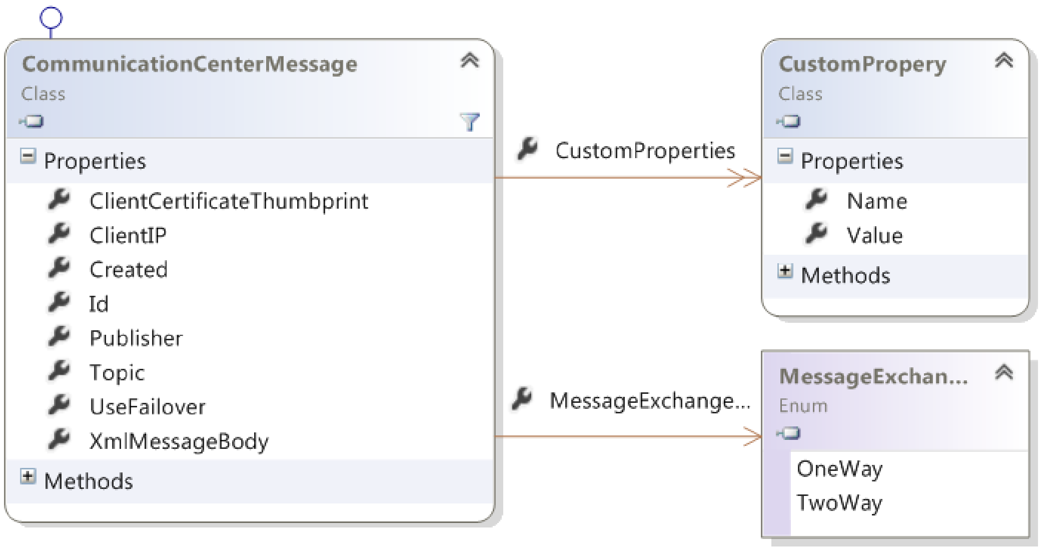


Figure 5.2: Internal Message

A Communication Center Message is serializable to XML, which means that it can easily be exchanged with SOAP Web services or stored in a queue or database. The payload is stored in the XML Message Body property, and it contains many metadata properties, which are used to add information to messages, like:

- **Id.** A unique identification of the message.
- **Created.** The time the message was created.

- **Topic.** The unique name of the topic to which the message has to be published.

- **Message Exchange Pattern.** Can be used to enable the synchronous request-response messaging extension.

- **Custom Properties.** Consist of a name and a value added to the internal messages constructed by the publishers. These properties can be used to add additional information to a message that can be used by all subscribers that receive the message.

Figure 5.3 depicts the part of the domain model that is used to perform message logging.

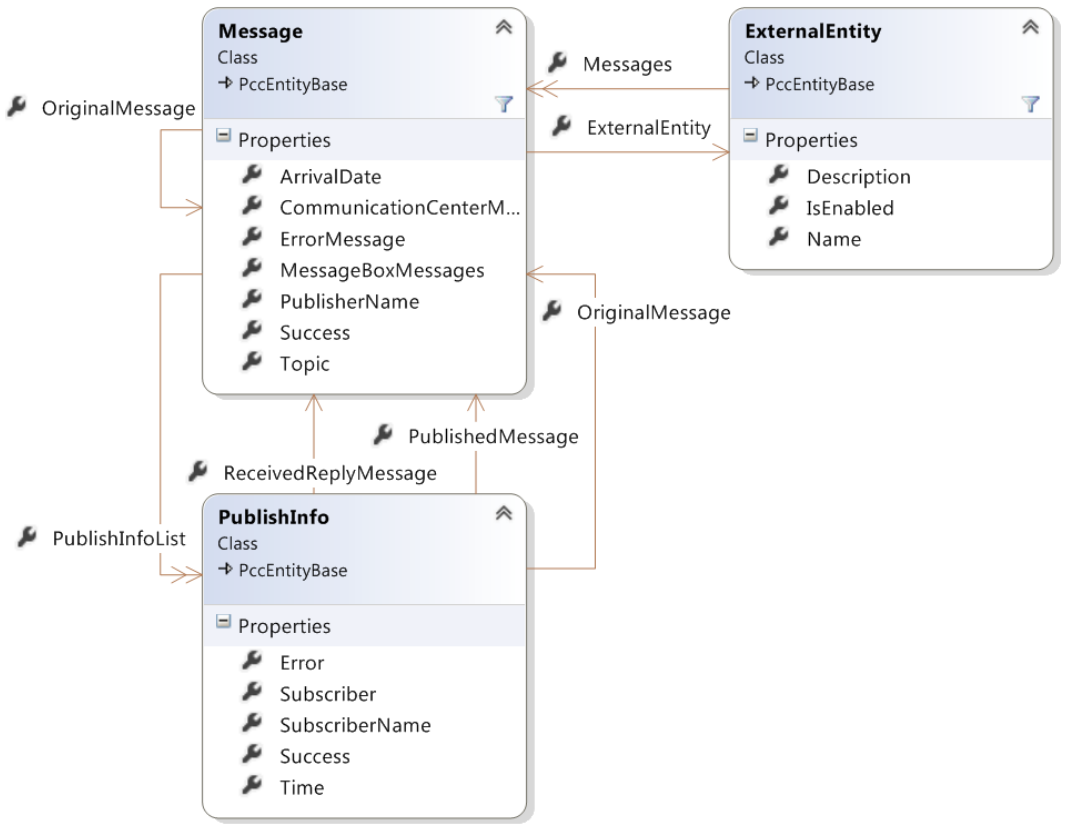


Figure 5.3: Message Flow Logging

- **Message.** If the publish-subscribe engine sends or receives an internal message, it creates an instance of the message class that contains additional information about the message and references to the other domain classes. It contains the time the message was received, the publisher of the message, the topic the message was published to and the previously described internal message with the actual content of the message.

- **Publish Info.** Each time a message is published to a subscriber, an instance of the publish info class is logged to the database. A publish info entity contains the message that was published, the time the message was published, the subscriber the message was published to and information indicating if the message was successfully published.

- **External Entity.** All incoming messages are linked to an external entity to provide security and usage information.

This model allows managers to use the management application to view the complete flow of a message through the runtime engine.

5.3 Retry Mechanism

If the publish-subscribe engine publishes a message to a subscriber that cannot successfully process the message, the subscriber can throw an exception. This causes the publish-subscribe engine to re-publish the message until the topic's maximum retry count has been reached.

Between each retry attempt, the publish-subscribe engine waits the amount of time that is specified by the topic's retry timeout value. If the topic's maximum retry count has been reached and the message still is not published successfully, the publish-subscribe engine re-publishes the message until the topic's maximum failover retry count has also been reached. However, this time the publish-subscribe engine sets the use fail over metadata property of the internal message to true before re-publishing the message.

Subscribers are responsible to check if this metadata property of the internal messages is set to true so they can take appropriate actions, such as, deliver the message to a fail over endpoint.

Subscribers can also throw a special exception indicating that the publish-subscribe engine does not need to re-publish this message. This can be used when performing a task that does not succeed by just retrying, such as, for example, XSD validation.

5.4 Request-Response Messaging

In addition to the default asynchronous publish-subscribe messaging it must also be possible to support synchronous request-response messaging, so we designed this extension for the publish-subscribe engine to provide this functionality.

This extension requires all subscribers to specify their supported MEP, which can be set to:

- **One-way.** Indicating that subscribers accept incoming messages, but they can not send a synchronous reply message. However, one-way subscribers can still publish messages.
- **Two-way.** Indicating that subscribers always send a synchronous reply message when they receive an incoming message.
- **None.** Indicating that subscribers do not accept any incoming messages, but they can still publish messages.

All publishers have to specify the MEP of the message, in addition to the unique name of a topic, when publishing a message. The MEP can be set to:

- **One-way.** Indicating that the publisher does not expect a reply message.
- **Two-way.** Indicating that the publisher expects a synchronous reply message.

This extension makes it possible for subscribers to receive a synchronous reply message when publishing a message to a topic, but at least one two-way subscriber has to be subscribed to the topic the message is published to, otherwise the publisher is notified that the message cannot be published successfully.

5.5 Chaining

The default behaviour of the publish-subscribe engine is to publish the same message to all subscribers of a topic in the order in which a manager specified the subscribers. However, there are situations in which one subscriber's output has to be the input of the next subscriber, and that is exactly the functionality this extension provides.

Figure 5.4 provides an example of a topic that has three subscribers, and is configured to use chaining. The numbers in Figure 5.4 identify a message, if a subscriber modifies a message the number is incremented to indicate the reply is a new message.

The message flow when chaining is enabled is as follows:

- The first subscriber receives the original message that was published to the topic.

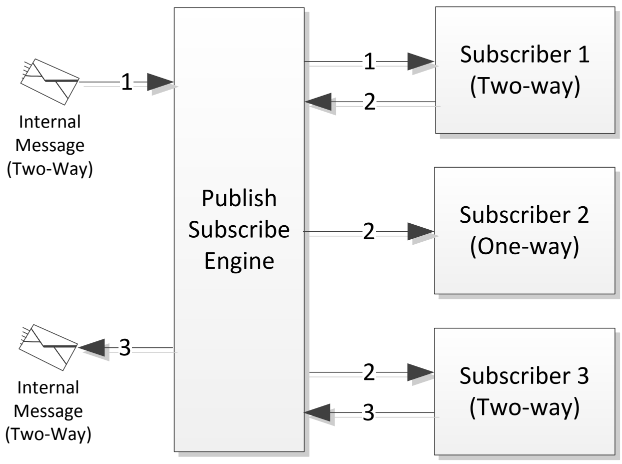


Figure 5.4: Internal Message

- The reply of a two-way subscriber is the input of the next subscriber.
- One-way subscribers do not modify messages, which means the same message the one-way subscriber received, is published to the next subscriber.

If chaining was not enabled all subscribers would receive the original message that was published to the topic (message number 1 in Figure 5.4).

The chaining extension allows the runtime engine to perform message processing using separate workflows, according to the VETO pattern [1] [2]. The VETO pattern is a common integration pattern that stands for Validate, Enrich, Transform, Operate (see Figure 5.5), and can ensure that consistent, validated data is routed through the bus.

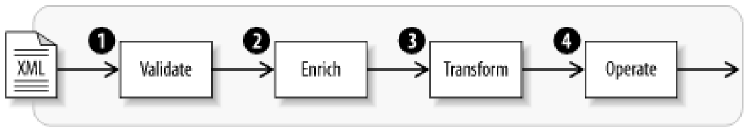


Figure 5.5: The VETO Pattern [1]

To do this developers can develop separate workflows to validate, enrich and transform messages. A manager can then subscribe these workflows to the same topic in the proper order, and enable chaining for this topic. The publish-subscribe engine then forwards the results between the workflows, and could forward the final result to an adapter.

5.6 Message Box

The runtime engine provides the message box extension to provide support for asynchronous messaging without external entities having to open inbound ports in their firewalls.

This extension allows the publish-subscribe engine to store the result of, for example, a long running business process implemented as a workflow in the publish-subscribe engine's message box. This allows the external entity that started the processing of the message to retrieve the result asynchronously.

Adapters are responsible to provide this functionality to the external entities because each adapter can provide the results in a different format and provide different behaviour for retrieving the results.

Examples of mechanisms for delivering messages to external entities are:

- Providing a Web service that can be polled to retrieve the messages in the message box.
- Storing the messages in a file on a FTP server.
- Periodically sending emails with the content of the messages.

The developer of an adapter is completely free to decide how to asynchronously deliver the messages to an external entity.

Figure 5.6 shows how the Message class from Figure 5.3 is used by the message box extension.

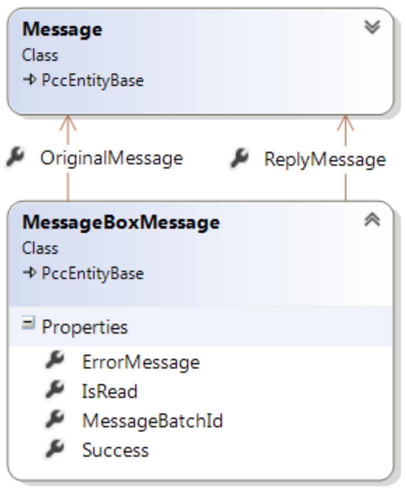


Figure 5.6: Message Box

Because all messages are logged, the MessageBoxMessage does not contain a copy of the actual message, but it is just refers to the following two logged messages:

- **Original Message.** The message that was sent by an external entity and initiated the long running process.
- **Reply Message.** The message that was received as a result of the long running process and has to be accessible via the message box.

In addition, it contains the following information for each message:

- **Success.** Indicates if the message was successfully published by the publish-subscribe engine.
- **Error Message.** Contains the error message if the message was not published successfully.
- **Is Read.** Indicates if the message has already been retrieved by an external entity.
- **Message Batch Id.** An unique identifier which can be used to mark a batch of messages as read with a single identifier. This functionality is demonstrated in Section 7.6.

5.7 Queues

Topics can be configured to use a queue to buffer incoming messages to cope with differing handling speeds and prevent message loss.

When a topic is configured to use the queuing extension, the publish-subscribe engine stores all messages that are published to the topic in a queue before they are published to the subscribers. Immediately after a message is stored in the queue, it returns a synchronous publish result indicating if the message was accepted and successfully stored.

The actual reply messages sent by the subscribers can be stored in the publish-subscribe engine's message box, which allows the publisher to receive these messages asynchronously.

Because the publish-subscribe engine only has to store the message in the queue and can then asynchronously process them, it can accept messages much faster than it processes them. However, the use of queues prevents synchronous messaging.

The publish-subscribe engine uses a configurable amount of threads to simultaneously process the queued messages. To prevent multiple threads from processing the same message, it automatically creates two queues for each topic that uses queues, one for incoming messages, and the other for messages that are currently being processed.

Before a thread starts processing a message, the message is moved to the currently processing queue for that topic, and when a thread finishes processing a message it is removed from the queue.

In case the runtime engine is unexpectedly restarted, this mechanism makes it possible to fully recover without sending unnecessary duplicated messages. This is possible because it knows which messages were being processed when it was interrupted, by looking in the currently processing queues. It can retrieve the logged publish information objects from the database for each of those messages. Based on these objects and the topic's retry settings, it is possible to determine how many times a message has already been published, and how many remaining times the message has to be published and to which subscribers.

5.8 Security

The runtime engine uses X.509 certificates for the authentication of external entities because this allows the existing customers' certificates to be reused. X.509 is, amongst other things, a standard for a public key infrastructure (PKI). However, the runtime engine can easily be extended to support additional security mechanisms.

Managers can use the management application to specify all external entities that have to be able to interact with the runtime engine. For each external entity, at least the name and the thumbprint of the external entity's certificate has to be specified. The thumbprint is a hash of the public key of the certificate, and is used to find the certificate in the certificate store of the computer running the runtime engine.

When an adapter publishes a message, it has to specify the thumbprint of the external entity in the metadata of the internal message that it publishes to the publish-subscribe engine. When the publish-subscribe engine receives a message, it uses this metadata to find the certificate.

If the external entity is not found, the message is rejected, otherwise the publish-subscribe engine uses the entities depicted in Figure 5.7 for authorization. The

SecurityInfo class represents a security rule indicating if an external entity can publish and/or receive messages to/from a topic or category.

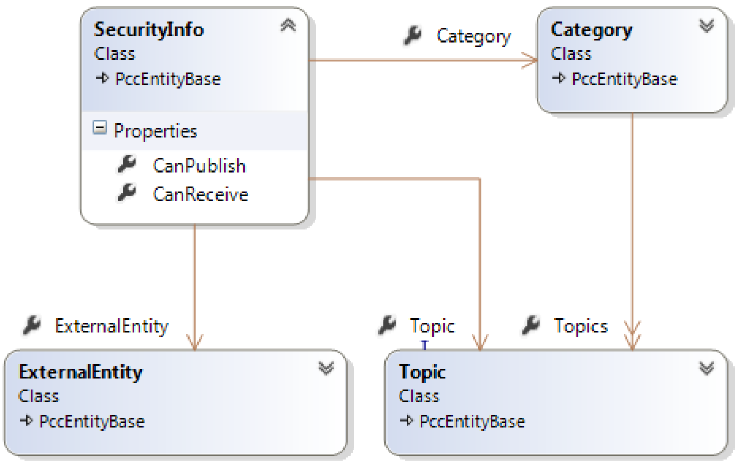


Figure 5.7: Security Model

5.9 Example Message Flow

Figure 5.8 depicts the message flow when a message is published to a topic that uses queuing, chaining and the message box extensions.

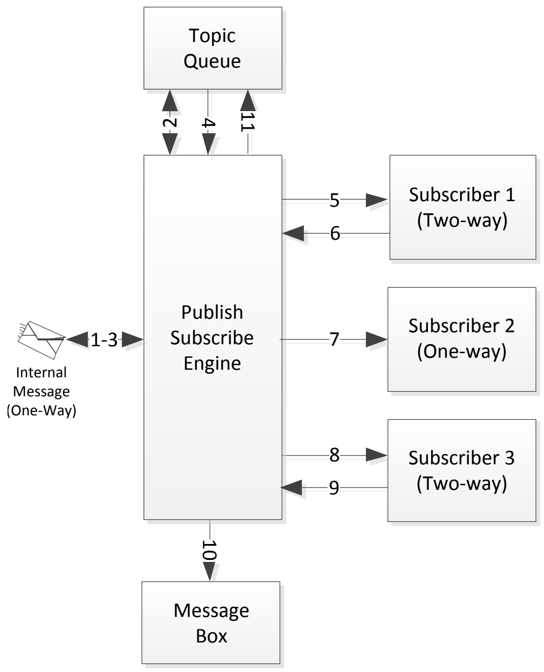


Figure 5.8: Message Publishing Process

Message flow when subscriber 1, 2 and 3 depicted in Figure 5.8 are all subscribed to the topic an example message is published to:

1. The publish-subscribe engine receives a message from a publisher and checks if the message can be accepted based on the topic's configuration and the thumbprint specified in the metadata of the internal message.
2. If the message is accepted the publish-subscribe engine stores it in the topic's queue.
3. The publish-subscribe engine returns a synchronous publish result to the publisher indicating if the message was accepted and successfully added to the queue.
4. When one of the message processing threads is available, the publish-subscribe engine retrieves a message from the topic's queue.
5. The publish-subscribe engine sends the retrieved message to subscriber 1.
6. The publish-subscribe engine receives the reply message from subscriber 1.
7. The publish-subscribe engine sends the reply message that was received from subscriber 1 to subscriber 2.
8. The publish-subscribe engine sends the reply message that was received from subscriber 1 to subscriber 3 because subscriber 2 is a one-way subscriber.
9. The publish-subscribe engine receives the reply message from subscriber 3.
10. The publish-subscribe engine stores the reply message that was received from subscriber 3 in the message box.
11. Once the publish-subscribe engine published the message successfully to all subscribers, it removes the message from the queue.

Figure 5.9 gives a complete view of the domain model.

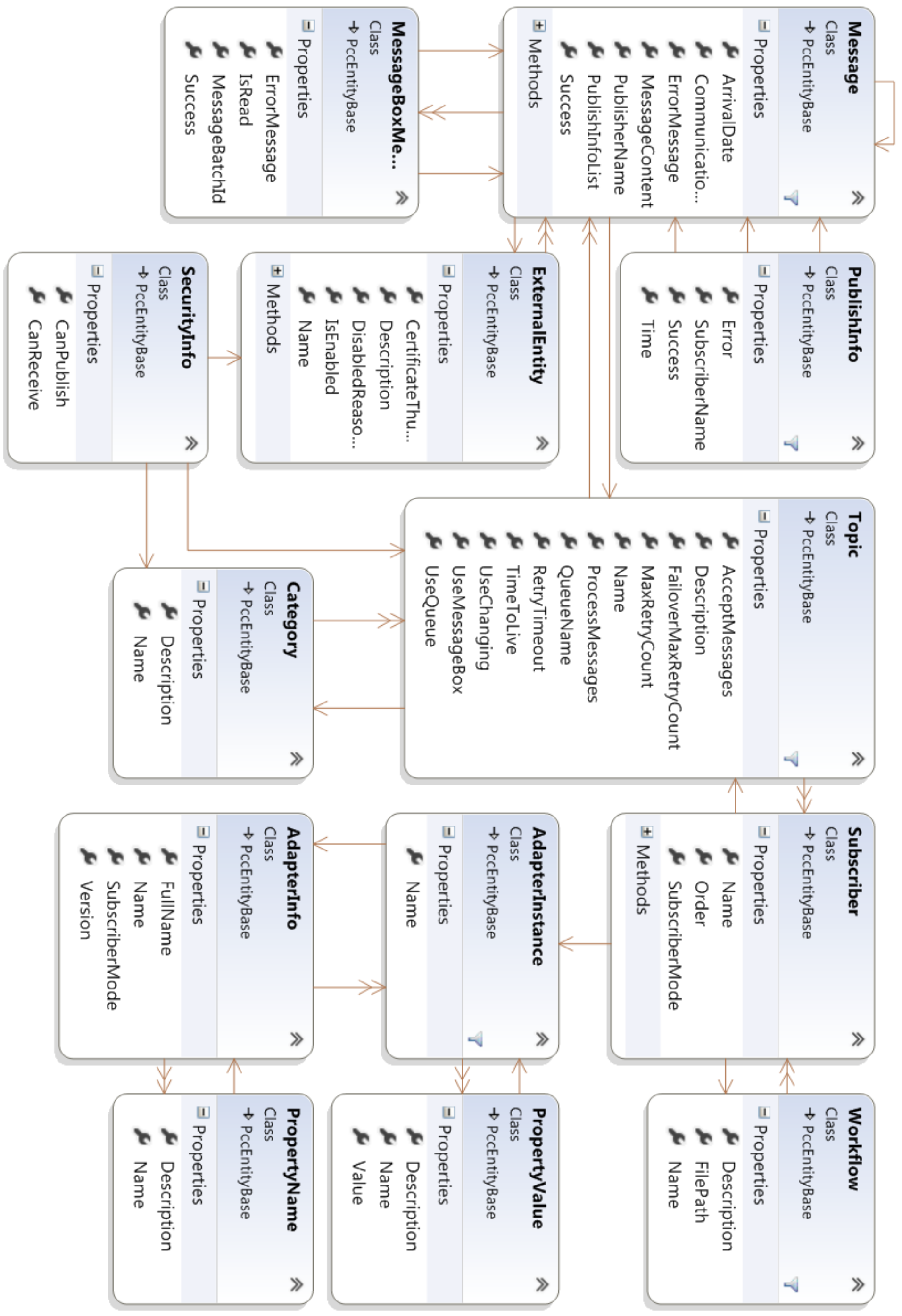


Figure 5.9: Domain Model

Chapter 6

Prototype Implementation

This chapter describes the implementation of the PCC2 prototype according to the architecture discussed in Chapter 4 and 5.

We chose to implement the prototype using the .NET framework to allow it to integrate with CompanyX's systems, but our architecture can be implemented using any implementation technologies.

This chapter is structured as follows:
Section 6.1 discusses the two most important technologies that we have identified for the development of custom ESB systems.
Section 6.2 describes the Service Host Factory that we have developed to facilitate the hosting of Web services.
Section 6.3 describes the communication with the management application.
Section 6.4 describes the implementation of the adapter framework.
Section 6.5 explains the implementation of the workflow framework.
Section 6.6 discusses the limitations of the implementation.

6.1 Implementation Technologies

Based on the knowledge acquired during our previous research we have identified the following two important technologies, which are part of the .NET Framework and can be used in the implementation of custom ESB systems, namely Windows Communication Foundation [22] [23] and 6.1.2 Windows Workflow Foundation [24].

6.1.1 Windows Communication Foundation

Windows Communication Foundation (WCF) is the most important building block part of the .NET framework that we have used to facilitate the interaction between the PCC2 and Web services is the. WCF can be used for building service-oriented applications and it offers a range of capabilities for communicating applications. The three most important aspects of WCF are:

1. Unification of the original .NET Framework communication technologies.

2. Interoperability with applications built on other technologies.

3. Explicit support for service-oriented development.

Because WCF unifies different approaches to communication, WCF implements Web services technologies defined by the WS-* specifications to allow more than just basic communication. Figure 6.1 shows how these specifications address several different areas, including basic messaging, security, reliability, transactions, and working with service metadata.

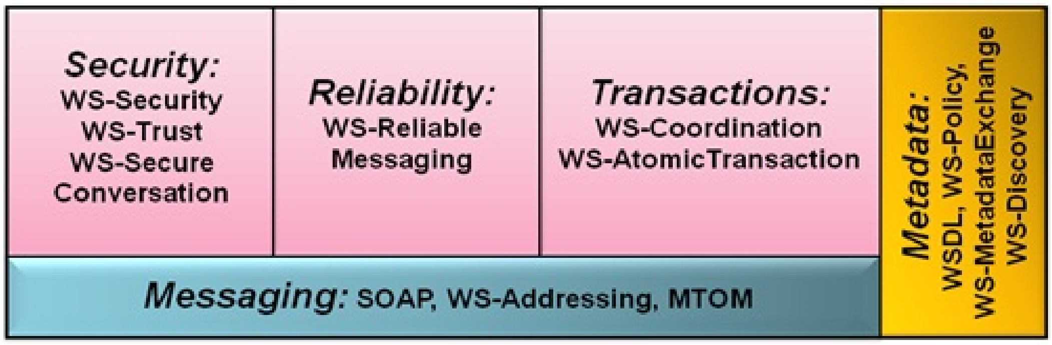


Figure 6.1: Web services standards implemented in WCF [22]

WCF facilitates the creation of distributed applications on Windows operating systems. It provides broad interoperability with other platforms because it implements SOAP and the most important WS-* specifications, along with RESTful communication. It also gives developers an environment for developing and deploying service-oriented applications because it offers explicit support for service-orientation.

Some benefits of WCF are:

1. It provides an extensive API which can be used to develop the base of an ESB.

2. Although it is also possible to implement an ESB without WCF, a compelling reason to use it is the amount of code that has to be written for basic infrastructure mechanisms (e.g., reliable messaging, transactions and security) is significantly reduced compared to using the traditional .NET communication mechanisms.

6.1.2 Windows Workflow Foundation

We have used Windows Workflow Foundation (WF), which is also part of the .NET Framework to implement the workflow framework.

We used WF in combination with WCF to provide the framework for defining workflows that can be called via Web services and updated at runtime to perform many tasks such as service composition, executing long-running business processes or message processing.

The primary goal of WF is to keep the applications logic unified, making it more understandable, while still letting the application scale [24].

WF provides a foundation for creating unified and scalable applications by supporting logic created using workflows. In addition, WF can also addresses other development challenges, such as coordinating parallel tasks and tracking program execution.

WF is a Microsoft technology that provides an API, an in-process workflow engine, and a rehostable designer to implement long-running processes as workflows within .NET applications.

The current version of WF was released as part of the .NET Framework version 4.5. We will refer to it as WF4.5 in this report.

6.2 Service Host Factory

When creating service-oriented applications, hosting Web services is a common task that has to be facilitated. The most flexible and easiest way to host Web services using WCF is to let an application host the services itself because that requires the least infrastructure to deploy. Hosting a service in this way is often referred to as "self-hosting".

To make the self-hosting of Web services as simple as possible we have developed a Service Host Factory that allows developers to host a Web service using different protocols and using different security features with a single method call. The Service Host Factory takes care of all functionality that is needed to host and configure the Web services. In addition, it allows developers to create a proxy that they can use to call the hosted Web service.

One challenge we encountered when implementing the Service Host Factory was that normal WCF services are self-contained classes without any external dependencies. That is why the default instantiation behavior for WCF is simply to call a parameter-less constructor of a service class.

However, a common task of the adapters instances used in the adapter framework is to host Web services, and the implementation of these Web services must have access to the adapter instance that hosted the Web service to access its properties.

Over 30 different extensibility points are available to allow developers to modify WCFs default behaviour. We used these extensibility points to allow the Service Host Factory to host services that require an adapter as a parameter when they are instantiated.

Figure 6.2 provides an overview of the Service Host Factory classes.

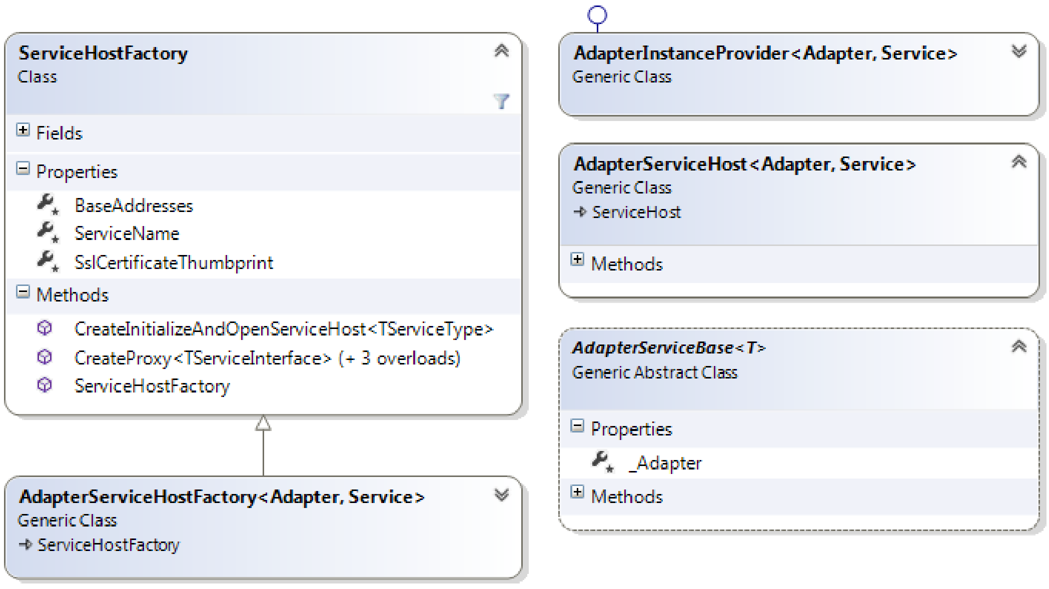


Figure 6.2: Service Host Factory classes

- **Service Host Factory.** Allows developers to self-host Web services that do not require any parameters when they are instantiated. To self host a Web service developers have to call the CreateInitializeAndOpenServiceHost method and specify the generic type of the class containing the service implementation. This method requires the following properties as parameters:
  - **Service Name.** The name of the service that is used to construct the endpoint address to reach the service.
  - **Base Addresses.** A list of base addresses that is used to determine the endpoint addresses of the endpoints that have to be hosted for the specified service. For example, if the following value is specified: "https://localhost: 1045, net.tcp://localhost:1046" the Service Host Factory hosts a HTTPS

and a TCP endpoint. The addresses of these endpoints are these base addresses suffixed with the service name.

  - **SSL Certificate Thumbprint.** Optionally, the thumbprint of the X.509 certificate that has to be used to secure the endpoints can be defined.

- **Adapter Service Host Factory.** Extends the Service Host Factory to add functionality that allows developers of adapters for the adapter framework to self-host Web services that need an adapter instance as a parameter.

- **Adapter Service Base.** Base class that service implementations need to extend to receive an adapter instance as a parameter.

- **Adapter Instance Provider.** Implements a WCF interface to override the default instantiation behavior of WCF to call the constructor of services that extend the AdapterServiceBase class instead of simply calling the parameterless constructor of the service class.

- **Adapter Service Host** Provides functionality that is required by WCF to actually host the service implementations as Web services.

6.3 Management Application

We chose to implement the management application with the .NET Framework 4.5 using Windows Forms (WinForms) [25], which is the name given to the graphical application programming interface in the Microsoft .NET Framework. This provides access to native Microsoft Windows interface elements by wrapping the Windows API in managed code.

The management applications features are covered in Chapter 8 by discussing its application to configure a complete integration scenario.

When the runtime engine starts it hosts a SOAP Web service using the Service Host Factory. This service is secured using X.509 certificates and allows multiple management applications to manage and monitor the runtime engine.

The runtime engine communicates with a database to store its configuration and log messages, but the management application does not directly communicate with this same database. This allows the database to run on a different server that is only accessible by the server running the runtime engine.

Data Transfer Objects

To transfer data between the management application and the runtime engine we use DTOs (Data Transfer Objects). DTO is a commonly used pattern in enterprise applications [26].A DTO is a simple container for a set of aggregated data that needs to be transferred across a process or network boundary. When designing a data transfer object, there are two primary choices: (1) use a generic collection or (2) create custom classes with explicit getter and setter methods. We chose to implement custom classes for each DTO because this provides strongly-typed objects that the management application can access exactly like any other class, so they provide compile-time checking. The main drawback is that it costs more development effort and is less flexible than using a generic collection.

6.4 Adapter Framework

We used the Managed Extensibility Framework (MEF) [27], which is a part of the .NET Framework to allow the adapter framework to dynamically load adapters from the file system at runtime. We chose MEF because it provides the level of flexibility we need, and it does not require any third-party components.

MEF can be used for creating lightweight, extensible applications and it allows application developers to discover and use extensions with no configuration required. It also lets extension developers easily encapsulate code and avoid fragile hard dependencies. MEF allows extensions to be reused within applications, but also across applications as well.

To develop an adapter for the adapter framework, a C# class has to be created that inherits from the abstract AdapterBase class (see Figure 6.3) and is exported using MEF.

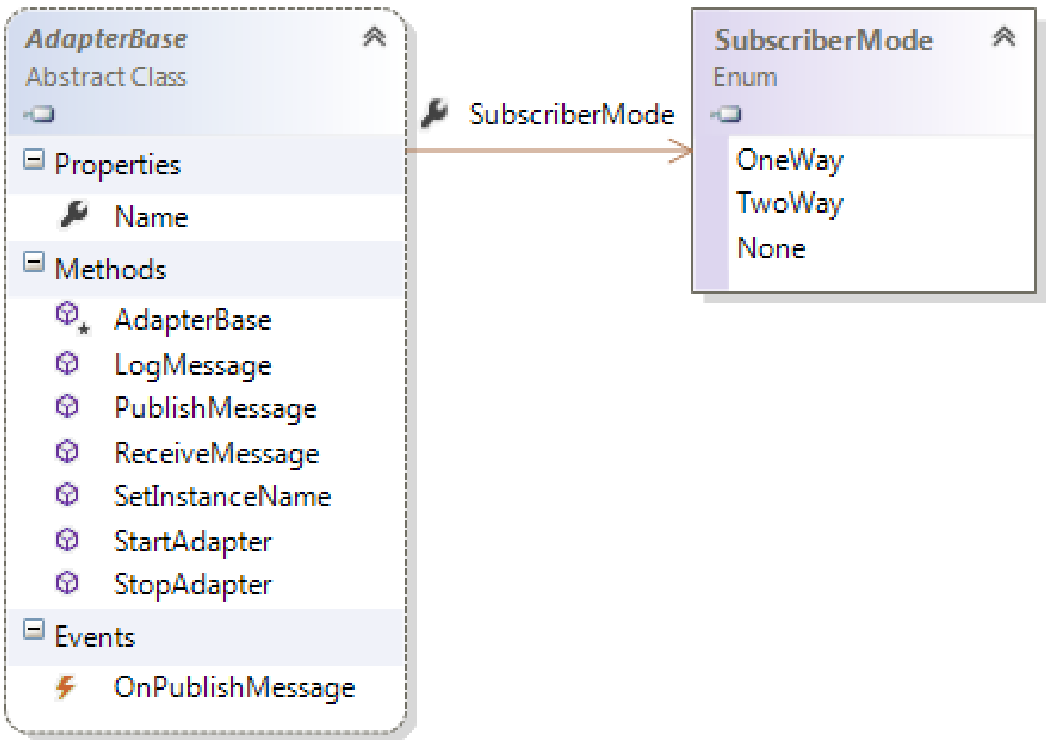


Figure 6.3: Adapters

The AdapterBase class contains methods that a concrete implementation can override to execute code when an adapter is started or stopped, and methods that take care of the communication with the Adapter Manager to publish or receive messages.

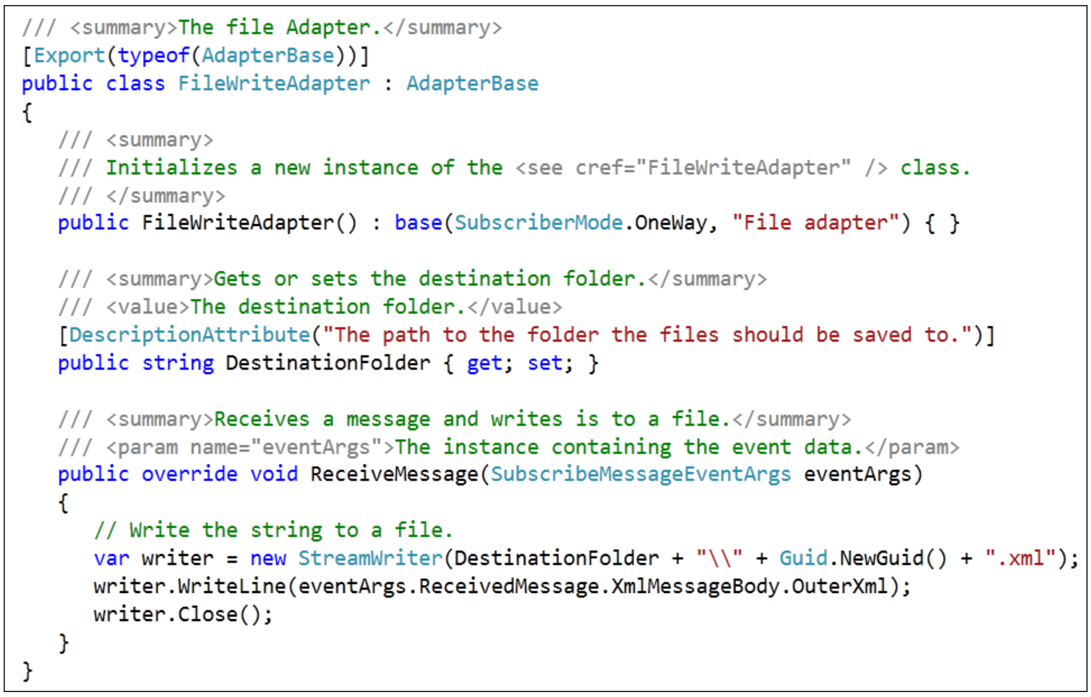
Figure 6.4 depicts the implementation code of an example adapter that can receive messages from the publish-subscribe engine and write them to a text file in a directory that is specified by a manager using the management application.

The constructor of the adapter specifies the adapters name and that it is a one-way subscriber, which means that the adapter does not return synchronous reply messages when it receives a message from the publish-subscribe engine.

The adapter declares one string property called DestinationFolder that can be set by a manager using the management application.

The ReceiveMessage method is used to inform the adapter that it received a message from the publish-subscribe engine.

To use a developed adapter such as the FileWriterAdapter, the adapter code has to be compiled to a dynamic-link library (DLL) and placed in the runtime engine's adapter folder. This causes the runtime engine to automatically discover the adapter and makes it possible to use the management application to register the adapter, create multiple instances of the registered adapter and configure the



```
/// <summary>The file Adapter.</summary>
[Export(typeof(AdapterBase))]
public class FileWriteAdapter : AdapterBase
{
   /// <summary>
   /// Initializes a new instance of the <see cref="FileWriteAdapter" /> class.
   /// </summary>
   public FileWriteAdapter() : base(SubscriberMode.OneWay, "File adapter") { }

   /// <summary>Gets or sets the destination folder.</summary>
   /// <value>The destination folder.</value>
   [DescriptionAttribute("The path to the folder the files should be saved to.")]
   public string DestinationFolder { get; set; }

   /// <summary>Receives a message and writes is to a file.</summary>
   /// <param name="eventArgs">The instance containing the event data.</param>
   public override void ReceiveMessage(SubscribeMessageEventArgs eventArgs)
   {
      // Write the string to a file.
      var writer = new StreamWriter(DestinationFolder + "\\" + Guid.NewGuid() + ".xml");
      writer.WriteLine(eventArgs.ReceivedMessage.XmlMessageBody.OuterXml);
      writer.Close();
   }
}
```

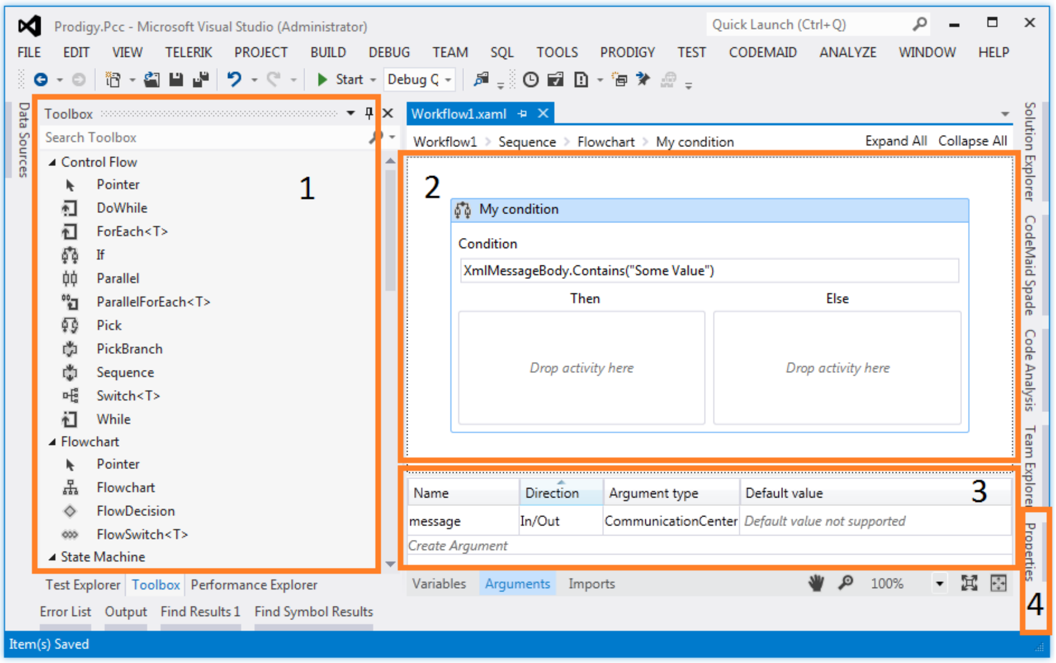
Figure 6.4: File Adapter

properties of each instance. Finally, the instances can be added as subscribers of topics. These steps are discussed in detail in Chapter 8.

6.5 Workflow Framework

The workflow framework allows developers to develop and deploy reusable WF 4.5 workflows that are are stored as Extensible Application Markup Language (XAML) files. XAML is a declarative XML-based language created by Microsoft that is used for initializing structured values and objects. XAML files can be created and edited with many applications, such as the Windows Workflow Foundation Designer in Visual Studio or with a standard text editor. Because workflows are stored as XAML files they can easily be modified at runtime without requiring any code to be recompiled.

Figure 6.5 depicts the Windows Workflow Foundation 4.5 Designer in Visual Studio 2012.



Condition

```
XmlMessageBody.Contains("Some Value")
```

Name	Direction	Argument type	Default value
message	In/Out	CommunicationCenter	*Default value not supported*
Create Argument			

Figure 6.5: Visual Studio 2012 - WF4.5 Designer

This interface has the following panes:

1. **Toolbox.** Displays activities that can be dragged onto the Canvas.

2. **Canvas.** Displays all activities that form the workflow.

3. **Arguments.** Displays the input parameters of the workflow.

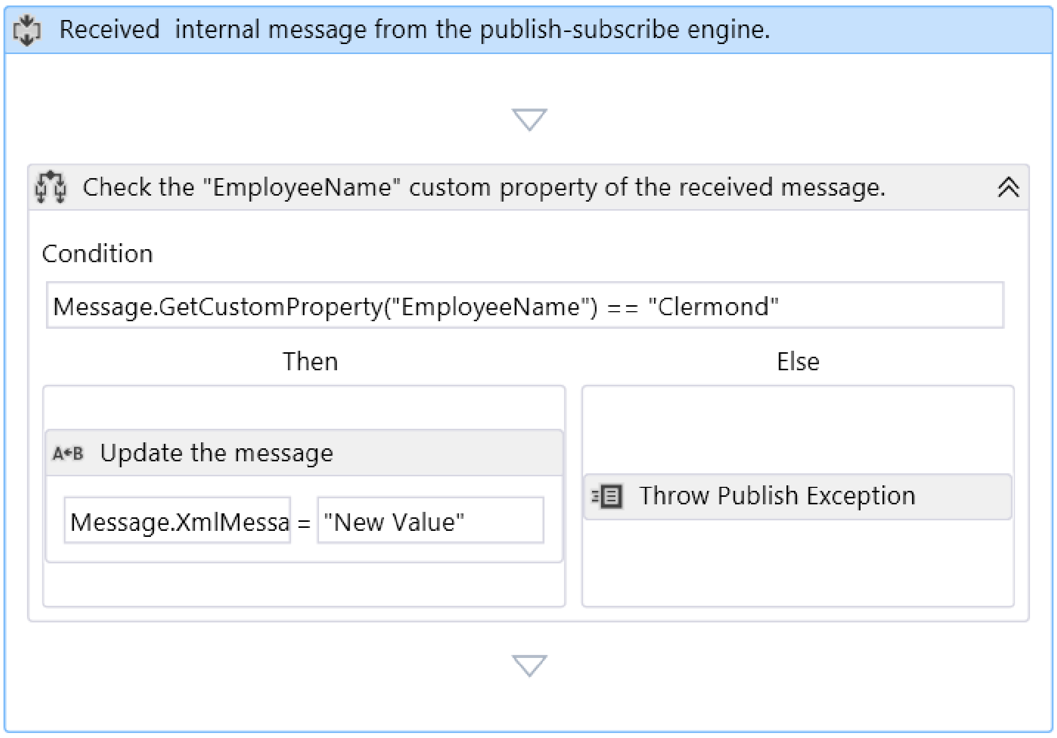
4. **Property Grid.** Can be used to configure properties of activities on the canvas.

WF 4.5 supports workflows that can have input parameters and these parameters can have different directions. The workflow framework allows the deployment of workflows that accept a single input parameter of the internal message (discussed in Section 5.2).

This parameter must have one of the following directions:

- **In.** This creates a one-way subscriber, which means that the workflow receives the internal message and can use it in the workflow, but it does not return a reply message to the publish-subscribe engine.

- **In/Out.** This creates a two-way subscriber that allows the workflow to modify the incoming message and return a new or modified message which can then be published to the next subscriber.


Received internal message from the publish-subscribe engine.

Check the "EmployeeName" custom property of the received message.

Condition

Message.GetCustomProperty("EmployeeName") == "Clermond"

Then

Update the message

Message.XmlMessa = "New Value"

Else

Throw Publish Exception

Figure 6.6: Visual Studio 2012 - WF4.5 Example

Figure 6.6 depicts a simple two-way example workflow that receives a message and transforms it by performing the following steps:

1. It checks if the value of the EmployeeName custom property of the internal message it received is equal to "Clermond".

2. If the condition is satisfied, it assigns a new value to a node of the received internal message that it found using a XPath expression. This is done by the assign activity in the "Then" branch of the If condition. The full text of the assign activity is: Message.XmlMessageBody.SelectSingleNode("XPath expression").InnerText = "New Value".

3. If the condition is not satisfied it throws a new publish exception that informs the publish-subscribe engine that further processing is not necessary.

6.6 Limitations

The prototype has been implemented according to the architecture discussed in Chapter 5and includes almost all of the described functionality, expect for the following two features:

- **Authorization** Referring to the authorization capabilities defined in Section 5.8 we have only implemented the authentication because this is a mandatory requirement (R7), but we have not implemented the authorization capabilities (Figure 5.7) because this is an optional requirements (R12) and we did not have time to implement it.

- **Recovery** Referring to the recovery capabilities in Section 5.7we have fully implemented the queuing functionality, but if the runtime engine is unexpectedly restarted it re-publishes all the messages that were being published as if they were received for the first time. Therefore, no messages are lost, but a subscriber of a topic with multiple subscribers might receive the same message multiple times. The recovery functionality without sending duplicated messages has not been implemented yet because it is not a requirement, and it is not necessary for our integration scenario.

Chapter 7

Case Implementation

This chapter we describes the implementation of the case study, which we performed to test, demonstrate and evaluate the functionality of the prototype.

This chapter is structured as follows:

Section 7.1 describes the integration scenario that we used in this case study.

Section 7.2 provides an overview of the different components that we have developed for this integration scenario.

Section 7.3 discusses an adapter that we have developed for this integration scenario which hosts Web services using the Service Host Factory.

Section 7.4 discusses an adapter that we have developed for this integration scenario which can send web requests.

Section 7.5 explains the implementation of a routing workflow that we have developed for this integration scenario to demonstrate the workflow framework.

Section 7.6 provides an overview of the flow of messages through the runtime engine when using the developed components.

7.1 Integration Scenario

CompanyX offer customers the service to digitally apply for export documents at the Chamber of Commerce (CoC) and print these documents at their own offices. This saves customers time and money because the CoC's prices for digitally submitted documents are less expensive than using hard copies, and in this way customers do not have to travel to the CoC.

An example of such a document is a Certificate of Origin document, which is used in the case of countries with which the EU has closed a trade agreement. This document grants lower import duties or even waives these entirely. There are many of such documents, and each document has to be sent to a different Web services of the CoC.

Figure 7.1 depicts the high-level components involved in this integration scenario.

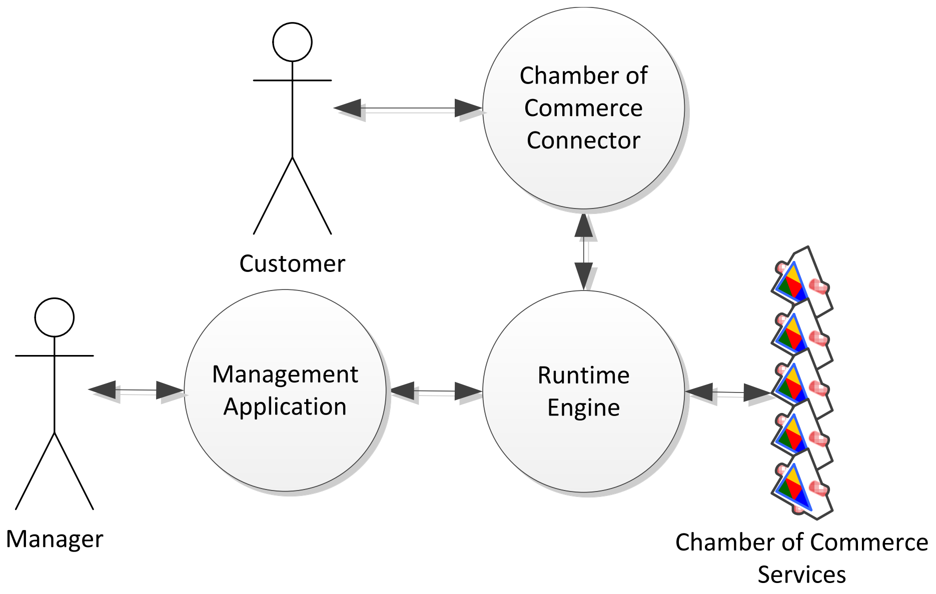


Figure 7.1: Integration Scenario Context Diagram

- **Chamber of Commerce Connector.** An application developed by CompanyX that is installed at the customers' computers to allow them to use a GUI to fill in export documents. Once a customer filled in an export document it is saved in an XML format that adheres to XSD schemas that are provided by the CoC.

  Each export document has its own message type, which is a string value indicating the type of the export document, and it can be used to determine to which CoC service the export document has to be sent.

- **Chamber of Commerce Services.** The services of the CoC that can receive digital export documents in XML format over an HTTPS connection.

- **Management Application.** The PCC2 management application that is used to manage the runtime engine.

- **Runtime Engine.** The PCC2 runtime engine that routes the messages to the proper CoC services based on their message type.

7.2 Overview

The PCC2 offers are many different approaches to provide a solution for this integration scenario. To demonstrate this, we have implemented two different solutions using two different approaches for this scenario. We call them the 'adapter approach' and the 'workflow approach'. We have chosen to provide two solutions that demonstrate the most important features of the implementation.

Figure 7.2 provides a an overview of the components that are used in both approaches. The difference between the two approaches is in the way the publish-subscribe engine publishes messages to its subscribers to route the messages to the proper CoC services. For each approach we have developed a Chamber of Commerce Connector Adapter (CoC adapter) for the communication with the CoC connector, and a HTTP Post adapter for the communication with the CoC services. These four adapters are discussed in detail in the sequel.

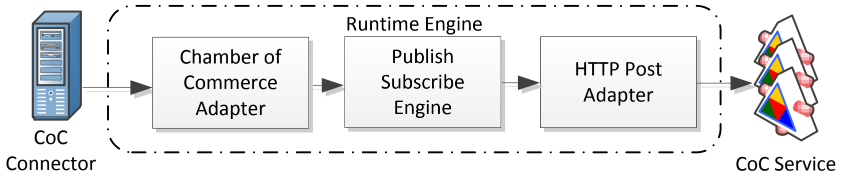


Figure 7.2: Message Flow Overview

Both solutions use the queuing extension, which prevents message loss in case the runtime engine is unexpectedly restarted and allows the publish-subscribe engine to process all messages asynchronously.

To prevent customers from having to open inbound ports in their firewalls to retrieve the asynchronous reply messages, we have modified the CoC connector to make use of the message box extension, instead of hosting a Web service to receive reply messages like when communicating with the PCC.

Finally, we have implemented a routing workflow that is used in the workflow approach to demonstrate the workflow framework.

Both approaches are discussed below.

7.2.1 Adapter Approach

The adapter approach demonstrates the strength of the adapter framework and publish-subscribe engine by providing a solution for the integration scenario without using workflows. Figure 7.3 depicts an overview of the message flow through the runtime engine when using the adapter approach.

The CoC adapter used in the adapter approach publishes messages of a different message type to a different topic. So, each topic represents a message type, and each topic has an instance of one of the HTTP Post adapters as its subscriber. Each instance of this HTTP Post adapter is configured with a single endpoint address of the proper CoC service.

Figure 7.3 only depicts three topics for readability, but there are actually twelve different message types, which means that using this approach twelve topics are necessary.

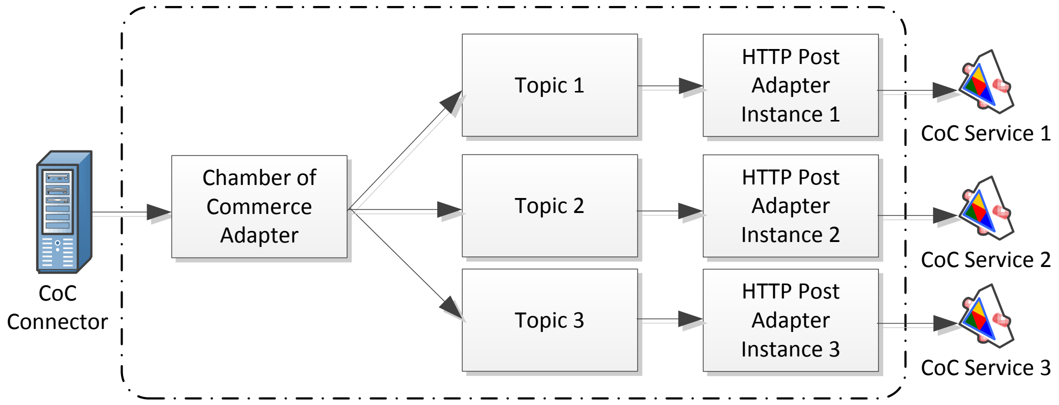

CoC Connector
Chamber of Commerce Adapter
Topic 1
Topic 2
Topic 3
HTTP Post Adapter Instance 1
HTTP Post Adapter Instance 2
HTTP Post Adapter Instance 3
CoC Service 1
CoC Service 2
CoC Service 3

Figure 7.3: Adapter Approach Overview

7.2.2 Workflow Approach

The workflow approach demonstrates the strength of the workflow framework and chaining extension by performing the routing of the messages in a workflow. Figure 7.4 depicts an overview of the message flow through the runtime engine when using the workflow approach. The difference with the adapter approach is that this variant of the CoC adapter publishes all messages of the different message types to a single topic, and this topic has two subscribers instead of one.

The first subscriber is a routing workflow that determines the proper CoC service to which the message has to be sent to, and enhances the received internal messages with this routing information by adding custom properties.

By enabling the chaining mechanism it is possible to use these enhanced messages as the input of the second subscriber. This is a variation of the HTTP Post Adapter, which sends the content of the enhanced internal messages to the endpoint address that was added by the routing workflow.

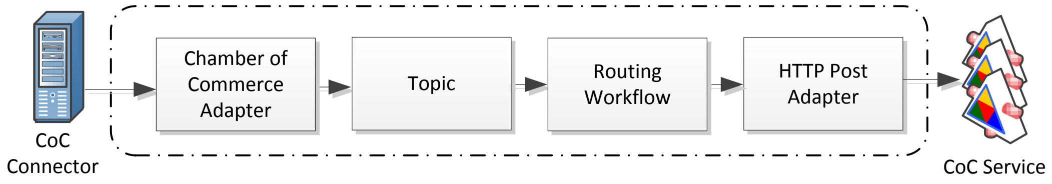


Figure 7.4: Workflow Approach Overview

7.3 Chamber of Commerce Connector Adapter

Figure 7.5 depicts the two CoC adapters that we have implemented. Both adapters use the None MEP, which means they cannot be a subscriber of a topic and do not receive any messages from the publish-subscribe engine, but they only publish messages to the publish-subscribe engine.

Each adapter host two SOAP Web services to communicate with the CoC connectors that are running on the customers' computers. To host these Web services both adapters, declare the properties that are required for the Service Host Factory, and declare the IsDownForMaintenace property that can be updated at runtime to accept or reject incoming messages.

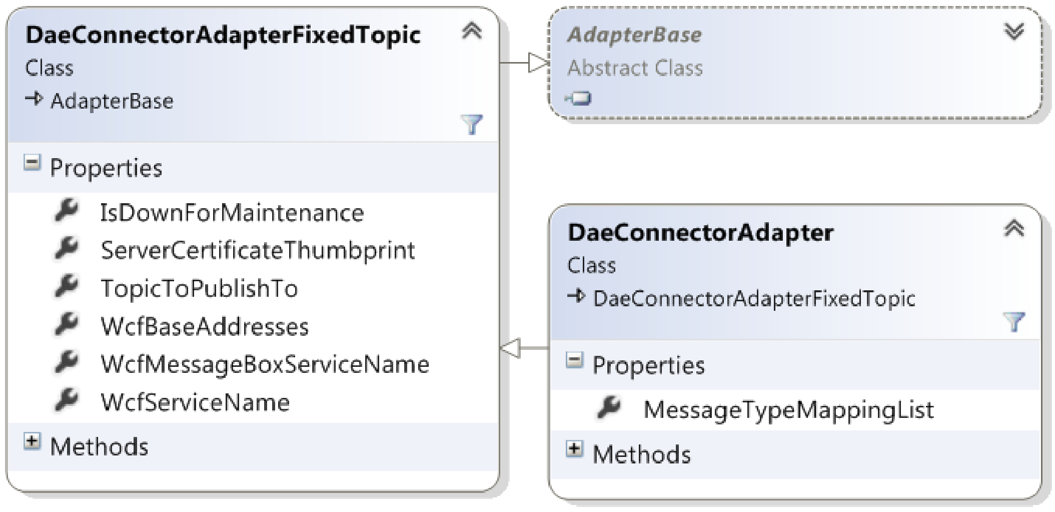


Figure 7.5: Chamber of Commerce Connector Adapters

- **Dae Connector Adapter.** This adapter is used in the adapter approach and provides the following additional property:
  - **Message Type Mapping List.** A list of strings containing the mapping between the message types and topic names that the adapter must use to determine the topic the incoming messages have to be published to.
- **Dae Connector Adapter Fixed Topic.** This adapter is used in the workflow approach and publishes all incoming messages to a single topic. In addition to the properties required by the Service Host Factory this adapter provides the following additional property:
  - **Topic To Publish To.** The name of the topic that all incoming messages have to be published to.

The difference between the two adapters is that the first adapter publishes all messages to a single topic, and the second adapter provides a property that allows managers to specify the mapping between the message type of a message and the topic the message has to be published to.

7.3.1 Web Service Implementation

Figure 7.6 depicts the two SOAP Web services that are hosted by the CoC adapter to communicate with the CoC connectors.

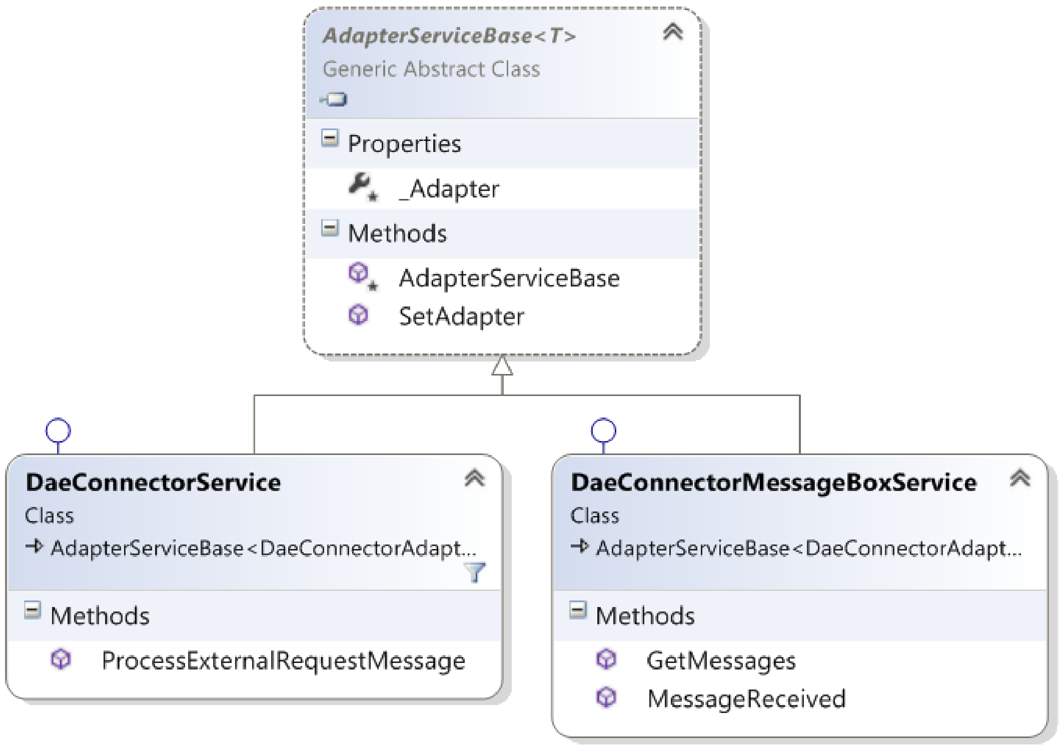


Figure 7.6: Web services

1. **Dae Connector Service.** This Web service receives messages containing the export documents in XML format that have to be sent to one of the CoC services, and it returns a synchronous reply to indicate if the publish-subscribe engine successfully accepted a message.

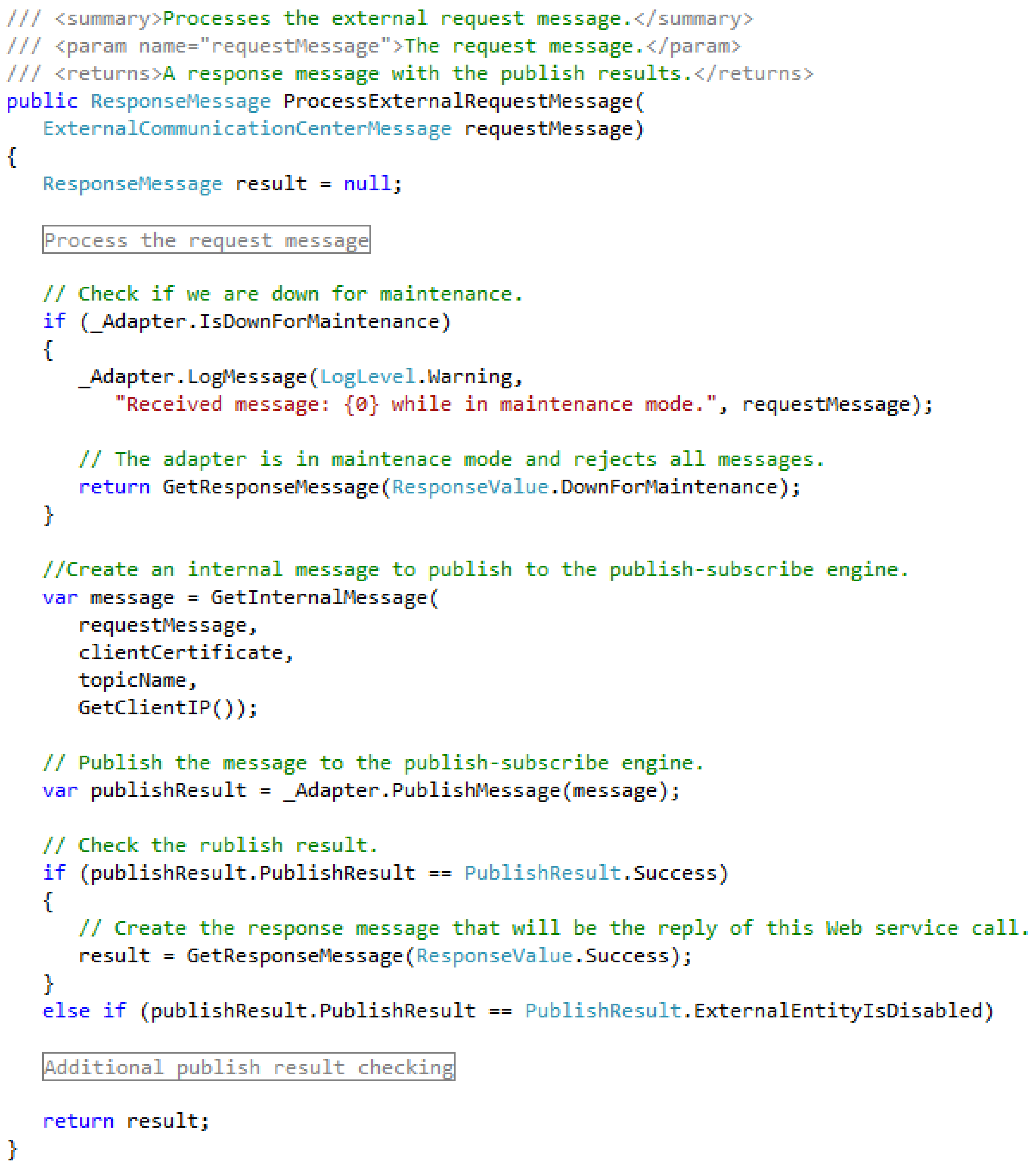
2. **Dae Connector Message Box Service.** This Web service allows customers to asynchronously retrieve the actual results which the publish-subscribe engine stores in the message box.

Figure 7.7 shows part of the C# code of the ProcessExternalRequestMessage method of the Dae Connector Service class that accepts the export documents in XML format.

When WCF receives a valid SOAP message, it automatically converts it to C# objects and invokes this method. The method processes the incoming message, and because this Web service is hosted with the Service Host Factory it can directly access the adapter instance that hosts this Web service.

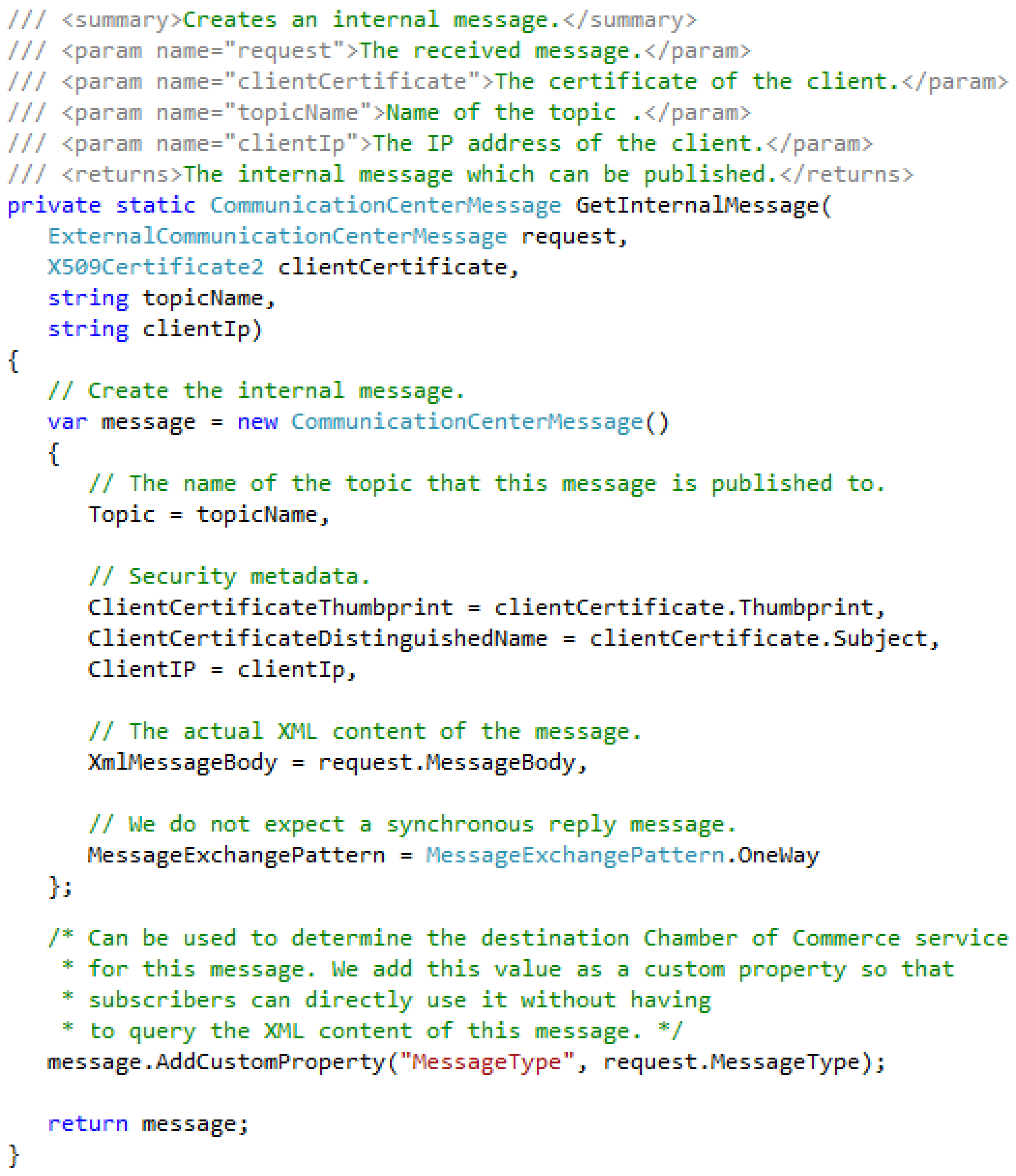
This method does the following:

- It checks the Is Down For Maintenance property of the adapter instance to determine if the adapter is in maintenance mode.
- If it is in maintenance mode the message is rejected, otherwise the GetInternalMessage helper method (showing in Figure 7.8) is called to create an internal message.
- The GetInternalMessage helper method creates a new internal message that can be published to the publish-subscribe engine. The created internal message's payload is the export document in XML format, and also contains metadata such as the thumbprint of the CoC connector's certificate, and a custom property that contains the message type of the received message.
- The created message is then published to the publish-subscribe engine, which returns a publish result indicating if the message was accepted.
- Finally, the publish result is checked and an appropriate reply is returned to the CoC connector as the result of the Web service call.



```
/// <summary>Processes the external request message.</summary>
/// <param name="requestMessage">The request message.</param>
/// <returns>A response message with the publish results.</returns>
public ResponseMessage ProcessExternalRequestMessage(
   ExternalCommunicationCenterMessage requestMessage)
{
   ResponseMessage result = null;

   Process the request message

   // Check if we are down for maintenance.
   if (_Adapter.IsDownForMaintenance)
   {
      _Adapter.LogMessage(LogLevel.Warning,
         "Received message: {0} while in maintenance mode.", requestMessage);

      // The adapter is in maintenace mode and rejects all messages.
      return GetResponseMessage(ResponseValue.DownForMaintenance);
   }

   //Create an internal message to publish to the publish-subscribe engine.
   var message = GetInternalMessage(
      requestMessage,
      clientCertificate,
      topicName,
      GetClientIP());

   // Publish the message to the publish-subscribe engine.
   var publishResult = _Adapter.PublishMessage(message);

   // Check the rublish result.
   if (publishResult.PublishResult == PublishResult.Success)
   {
      // Create the response message that will be the reply of this Web service call.
      result = GetResponseMessage(ResponseValue.Success);
   }
   else if (publishResult.PublishResult == PublishResult.ExternalEntityIsDisabled)

   Additional publish result checking

   return result;
}
```

Figure 7.7: CoC Adapter - Web Service Implementation



```
/// <summary>Creates an internal message.</summary>
/// <param name="request">The received message.</param>
/// <param name="clientCertificate">The certificate of the client.</param>
/// <param name="topicName">Name of the topic .</param>
/// <param name="clientIp">The IP address of the client.</param>
/// <returns>The internal message which can be published.</returns>
private static CommunicationCenterMessage GetInternalMessage(
   ExternalCommunicationCenterMessage request,
   X509Certificate2 clientCertificate,
   string topicName,
   string clientIp)
{
   // Create the internal message.
   var message = new CommunicationCenterMessage()
   {
      // The name of the topic that this message is published to.
      Topic = topicName,

      // Security metadata.
      ClientCertificateThumbprint = clientCertificate.Thumbprint,
      ClientCertificateDistinguishedName = clientCertificate.Subject,
      ClientIP = clientIp,

      // The actual XML content of the message.
      XmlMessageBody = request.MessageBody,

      // We do not expect a synchronous reply message.
      MessageExchangePattern = MessageExchangePattern.OneWay
   };

   /* Can be used to determine the destination Chamber of Commerce service
    * for this message. We add this value as a custom property so that
    * subscribers can directly use it without having
    * to query the XML content of this message. */
   message.AddCustomProperty("MessageType", request.MessageType);

   return message;
}
```

Figure 7.8: CoC Adapter - Web service Implementation

7.4 HTTP Post Adapter

Figure 7.9 depicts the two HTTP Post adapters that we have implemented. The HTTP Post adapter can send messages to an endpoint using the HTTP and HTTPS Post method. Both adapters use two-way MEP, which means that they can be a subscriber of a topic to receive messages from the publish-subscribe engine and send synchronous reply messages with the results returned by the endpoints.

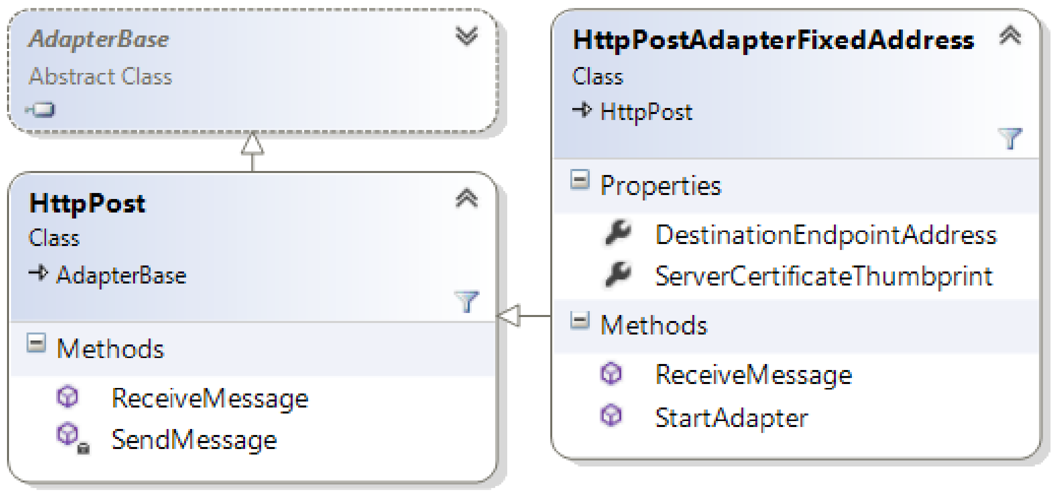


Figure 7.9: HTTP(S) Post Adapters

- **Http Post Adapter Fixed Address.** This adapter is used in the adapter approach and declares the following two properties that have to be configured for each instance of the adapter:
  - **Destination Endpoint Address.** The endpoint address to which the adapter instance has to send all messages.
  - **Server Certificate Thumbprint.** The thumbprint of the certificate that has to be used to secure the connection. If this value is set the adapter uses HTTPS, otherwise HTTP is used.
- **Http Post Adapter.** This adapter is used by the workflow approach and it declares no properties, but requires the same two values to be passed as custom properties of the internal messages it receives from the publish-subscribe engine.

So, the main difference between the two adapters is that the first adapter requires a new instance of the adapter for each different endpoint, and the second adapter only needs a single instance, but requires that the internal messages it receives contain the two properties as custom properties.

7.5 Route Workflow

Figure 7.10 depicts the routing workflow in the WF 4.5 Designer in Visual Studio 2012 that enhances internal message with the two custom properties the HTTP Post adapter requires. It does this based on the MessageType custom property that is added by the CoC adapter in the method GetInternalMessage (see Figure 7.8).

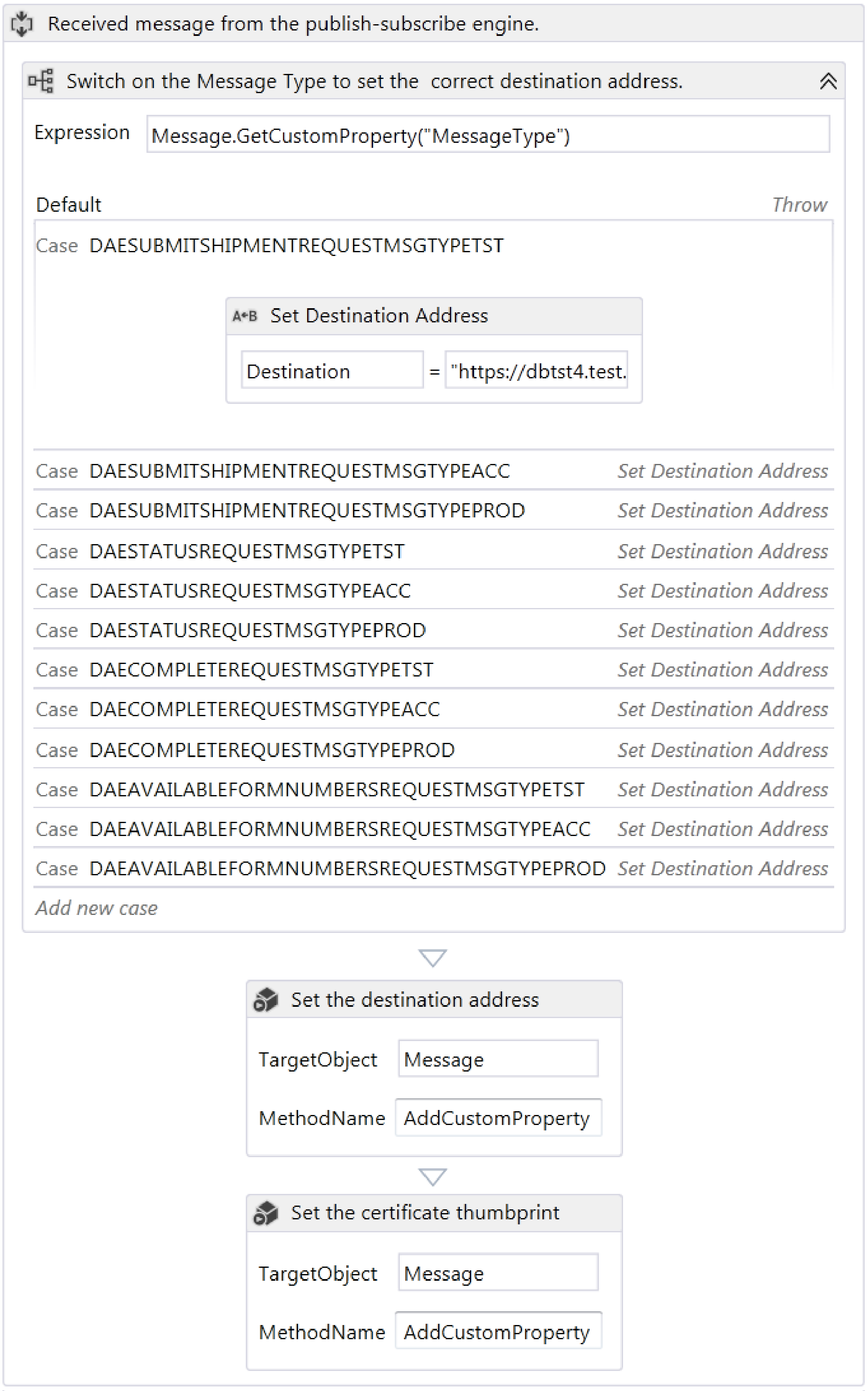
The workflow consists of a sequence activity that contains the following three sub-activities:

1. A switch statement on the custom property of the internal message that contains the message type. Each case statement contains an assign activity which assigns the destination address of the correct CoC service for the specific message type to the Destination variable. For readability only one case statement is expanded in Figure 7.10, the others are collapsed.

   The Default switch statement throws a publish exception indicating that the message type is unknown, which prevents the publish-subscribe engine from publishing the message to the next subscriber.

2. An activity that calls the AddCustomProperty method of the internal message to add the value of the Destination variable as a custom property.

3. An activity that calls the AddCustomFProperty method of the internal message to add the thumbprint of the certificate that has to be used for the communication with the CoC services as a custom property.



Received message from the publish-subscribe engine.

Switch on the Message Type to set the correct destination address.

Expression Message.GetCustomProperty("MessageType")

Default Throw

Case DAESUBMITSHIPMENTREQUESTMSGTYPETST

Set Destination Address

Destination = "https://dbtst4.test.

Case	DAESUBMITSHIPMENTREQUESTMSGTYPEACC	Set Destination Address
Case	DAESUBMITSHIPMENTREQUESTMSGTYPEPROD	Set Destination Address
Case	DAESTATUSREQUESTMSGTYPETST	Set Destination Address
Case	DAESTATUSREQUESTMSGTYPEACC	Set Destination Address
Case	DAESTATUSREQUESTMSGTYPEPROD	Set Destination Address
Case	DAECOMPLETEREQUESTMSGTYPETST	Set Destination Address
Case	DAECOMPLETEREQUESTMSGTYPEACC	Set Destination Address
Case	DAECOMPLETEREQUESTMSGTYPEPROD	Set Destination Address
Case	DAEAVAILABLEFORMNUMBERSREQUESTMSGTYPETST	Set Destination Address
Case	DAEAVAILABLEFORMNUMBERSREQUESTMSGTYPEACC	Set Destination Address
Case	DAEAVAILABLEFORMNUMBERSREQUESTMSGTYPEPROD	Set Destination Address

Add new case

Set the destination address

TargetObject Message

MethodName AddCustomProperty

Set the certificate thumbprint

TargetObject Message

MethodName AddCustomProperty

Figure 7.10: Visual Studio 2012 - WF4.5 Designer

7.6 Message Flow

Figure 7.11 depicts the flow of a message through the runtime engine when using the adapter approach.

Figure 7.12 depicts the detailed flow when using the workflow approach.

This section only discusses the flow of the workflow approach in detail because two approaches are quite similar, and the differences can clearly be seen by comparing Figure 7.11 and 7.12.

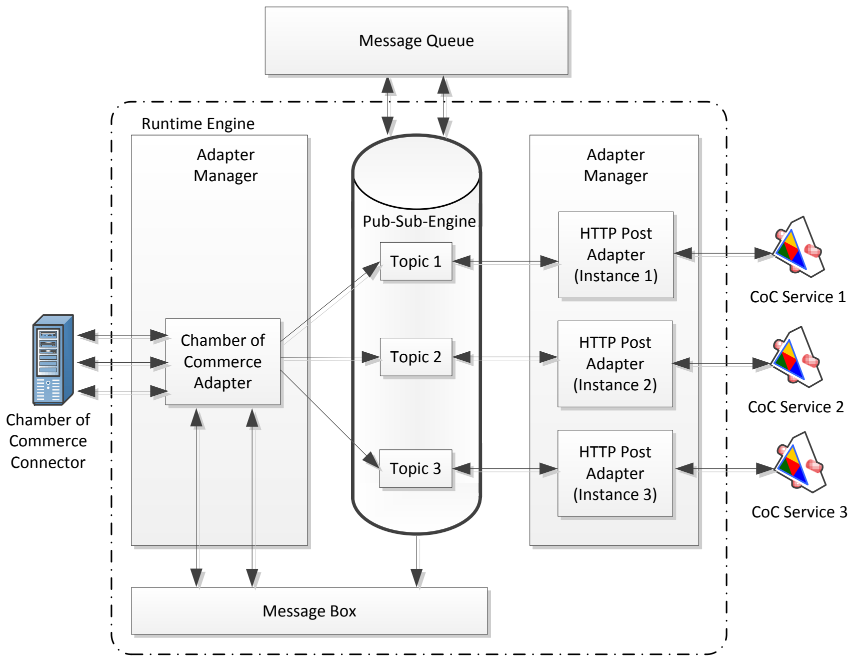

Message Queue
Runtime Engine
Adapter Manager
Pub-Sub-Engine
Topic 1
Topic 2
Topic 3
Adapter Manager
HTTP Post Adapter (Instance 1)
HTTP Post Adapter (Instance 2)
HTTP Post Adapter (Instance 3)
CoC Service 1
CoC Service 2
CoC Service 3
Chamber of Commerce Adapter
Chamber of Commerce Connector
Message Box

Figure 7.11: Adapter Approach Message Flow

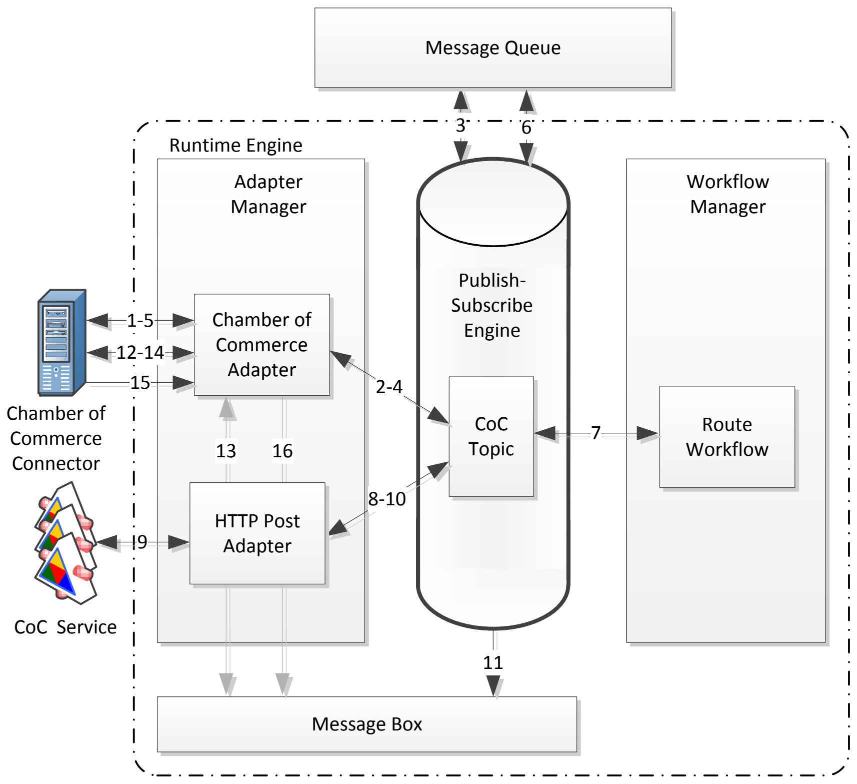


Figure 7.12: Workflow Approach Message Flow

The flow is the following:

1. A CoC connector sends an export document in XML format to the CoC adapter using the SOAP Web service that the adapter hosts to accept export documents. While the connection with the CoC adapter stays open, steps 2, 3 and 4 are performed.

2. The CoC adapter wraps the received XML export document in an internal message, adds the thumbprint of the certificate that was used to establish the connection with the adapter's Web service as metadata to the created internal message, and then publishes the internal message to the publish-subscribe engine.

3. The publish-subscribe engine inspects the published message and verifies that the sender of the message is known based on the provided thumbprint. If the sender is known, the publish-subscribe engine retrieves the topic from the database. Because the topic is configured to use the queuing extension, the publish-subscribe engine stores all messages in the queue that is specified in the topics settings.

4. Because the topic uses the queuing mechanism, the publish-subscribe engine

immediately returns a publish result indicating if the message has been accepted and successfully stored in the queue.

5. The CoC adapter retrieves the publish result from the publish-subscribe engine and uses the open connection to notify the CoC connector if the message was accepted, and then the connection is closed.

6. When one of the message processing threads is available, the publish-subscribe engine retrieves a message from the topic's queue.

7. The publish-subscribe engine inspects the retrieved message and publishes the message to the first subscriber, which is the routing workflow.

8. The publish-subscribe engine publishes the enhanced internal message that the routing workflow returned to the HTTP Post adapter, which is possible because we have configured the topic to use the chaining mechanism.

9. The HTTP Post adapter sends the content of the internal message, which is the export document in XML format, to the proper CoC service by using the endpoint address that is specified in the metadata of the enhanced internal message.

10. Because the HTTP Post adapter is a two-way subscriber, it returns the reply it received from the CoC service to the publish-subscribe engine.

11. Because the HTTP Post adapter is the last subscriber and chaining is enabled, the publish-subscribe engine stores the reply it received from the HTTP Post adapter in the message box. The message processing thread then removes the message from the queue and terminates because it completed publishing the message.

12. The CoC connector periodically polls the message box Web service hosted by the CoC adapter to check if there are results for messages that were accepted by the publish-subscribe engine.

13. The CoC adapter leaves the connection with the CoC adapter open, and queries the message box table in the database to retrieve all unread messages for the proper CoC connector. It does this by using the thumbprint of the certificate that is used to establish the connection with the message box Web service to filter the messages.

The CoC adapter also assigns a unique batch id to all the unread messages, and updates the message box table in the database.

14. The CoC adapter uses the open connection to return the retrieved messages together with the unique batch id to the CoC connector. If the publish-subscribe engine fails to process any message for this CoC connector, it returns the errors that occurred instead of actual reply message, and then the connection is closed.
15. If the CoC connector receives all reply messages successfully, it calls one last Web service method of the message box Web service, and passes the unique batch id it received to indicate that it has successfully received all messages with the specified batch id.
16. The CoC adapter marks all messages with the specified batch id as read, which prevents them from being returned the next time the CoC connector polls the message box Web service for unread messages.

Chapter 8

Case Configuration

This chapter demonstrates the operation of the management application to configure the runtime engine according to the approaches discussed in Chapter 7 by using the developed workflow and adapters. The chapter also discusses the workflow approach in detail because it covers most of features of the the management application. The chapter also briefly discusses the configuration of the adapter approach to cover the remaining features of the management application.

This chapter is structured according to the management applications main menu which is depicted in Figure 8.1.

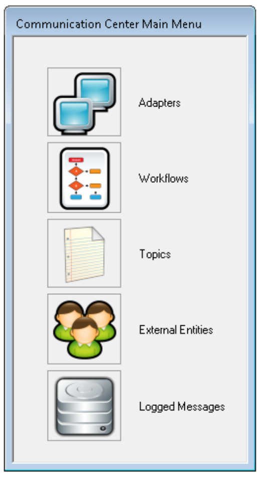
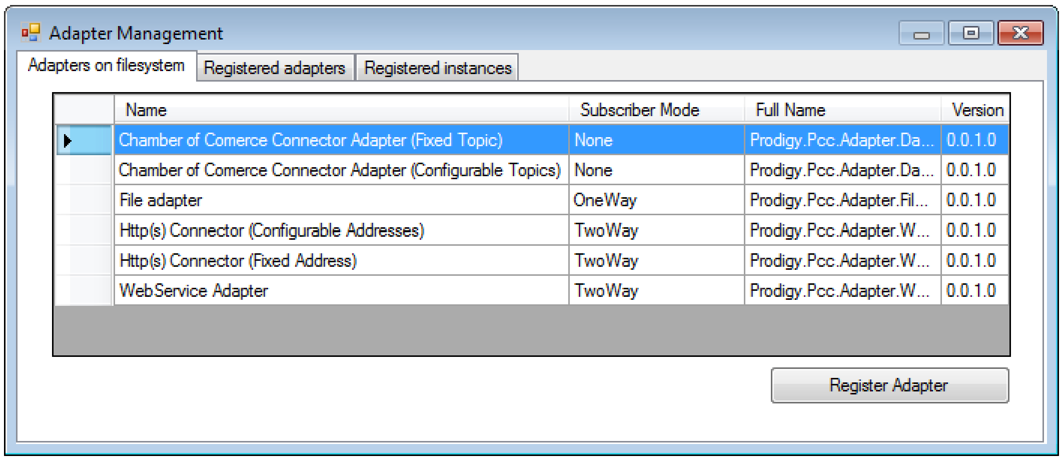


Figure 8.1: PCC2 Main Menu

Section 8.1 describes how developed adapters can be registered and instantiated.
Section 8.2 explains how a routing workflow can be registered.
Section 8.3 discusses the interface to manage topics and categories.
Section 8.4 explains how to specify external entities that are allowed to communicate with the publish-subscribe engine.
Section 8.5 shows how the complete flow of a message through the runtime engine can be viewed.
Section 8.6 explains how the runtime engine can be configured to use the adapter approach instead of the workflow approach.

8.1 Adapters

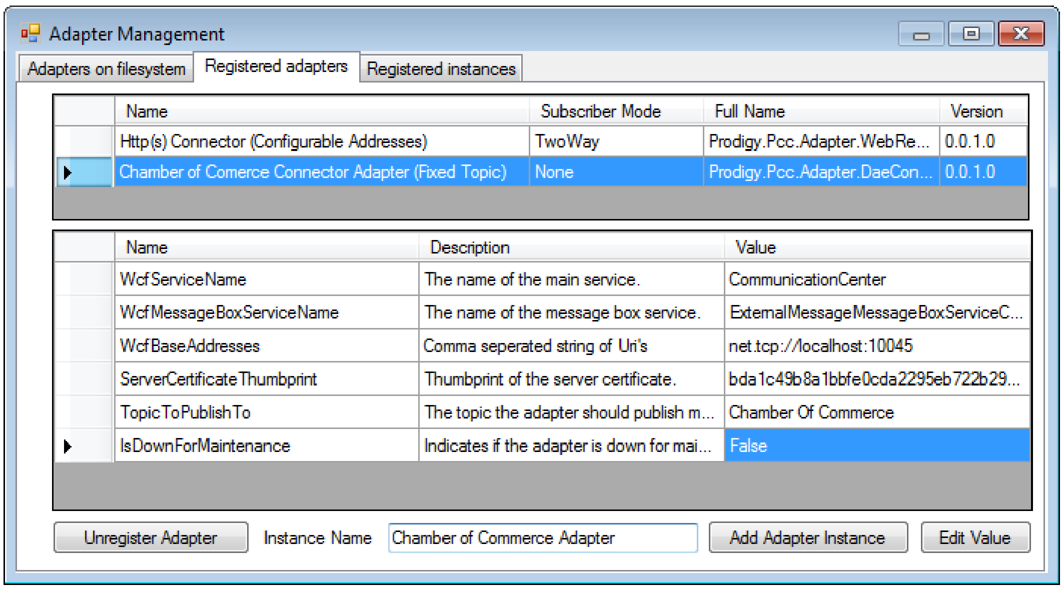
Figure 8.2 depicts the interface that lists the adapters which we have developed, compiled to DLL files and placed in the runtime engines' adapter folder. For the workflow approach we have registered the intended versions of the CoC adapter and the HTTP Post adapter by selecting them and clicking the Register Adapter button.



Name	Subscriber Mode	Full Name	Version
Chamber of Comerce Connector Adapter (Fixed Topic)	None	Prodigy.Pcc.Adapter.Da...	0.0.1.0
Chamber of Comerce Connector Adapter (Configurable Topics)	None	Prodigy.Pcc.Adapter.Da...	0.0.1.0
File adapter	OneWay	Prodigy.Pcc.Adapter.Fil...	0.0.1.0
Http(s) Connector (Configurable Addresses)	TwoWay	Prodigy.Pcc.Adapter.W...	0.0.1.0
Http(s) Connector (Fixed Address)	TwoWay	Prodigy.Pcc.Adapter.W...	0.0.1.0
WebService Adapter	TwoWay	Prodigy.Pcc.Adapter.W...	0.0.1.0

Figure 8.2: Loaded Adapters

Once the adapters have been registered, the interface depicted in Figure 8.2 can be used to create instances of the adapters by specifying a name for the instance, values for the declared properties and then clicking the Add Adapter Instance button. We assign the value "Chamber of Commerce" to the TopicToPublishTo property, to configure the adapter to publish all messages to this topic.



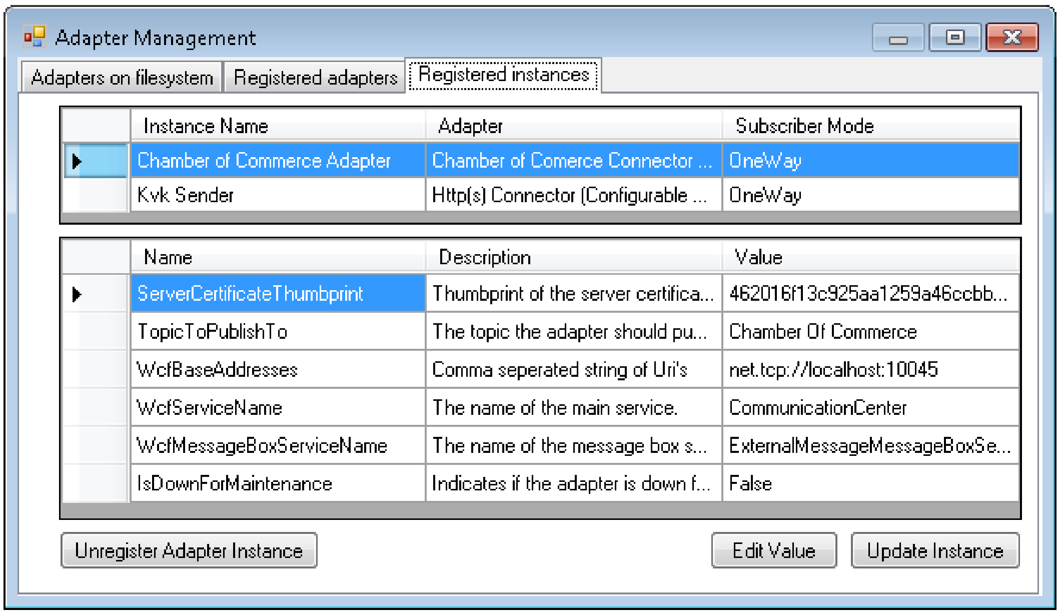
Name	Subscriber Mode	Full Name	Version
Http(s) Connector (Configurable Addresses)	TwoWay	Prodigy.Pcc.Adapter.WebRe...	0.0.1.0
Chamber of Comerce Connector Adapter (Fixed Topic)	None	Prodigy.Pcc.Adapter.DaeCon...	0.0.1.0

Name	Description	Value
WcfServiceName	The name of the main service.	CommunicationCenter
WcfMessageBoxServiceName	The name of the message box service.	ExternalMessageMessageBoxServiceC...
WcfBaseAddresses	Comma seperated string of Uri's	net.tcp://localhost:10045
ServerCertificateThumbprint	Thumbprint of the server certificate.	bda1c49b8a1bbfe0cda2295eb722b29...
TopicToPublishTo	The topic the adapter should publish m...	Chamber Of Commerce
IsDownForMaintenance	Indicates if the adapter is down for mai...	False

Instance Name: Chamber of Commerce Adapter

Figure 8.3: Registered Adapters

Once the Add Adapter Instance button is pressing, the management application sends the information to the runtime engine, which adds the new instance to the database. The runtime engine then instantiates a new instance of the adapter class in a separate thread, and sets the properties with the specified values. The adapter instance is then ready to perform its task and can be used as a subscriber of a topic, if it does not use 'None' as its MEP. The properties of the running adapter instances can also be viewed and updated at runtime using the interface depicted in Figure 8.4.



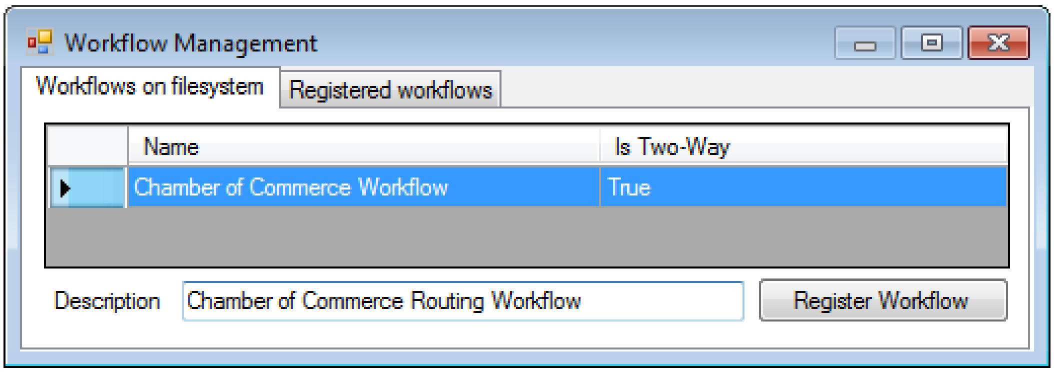
Instance Name	Adapter	Subscriber Mode
Chamber of Commerce Adapter	Chamber of Comerce Connector ...	OneWay
Kvk Sender	Http(s) Connector (Configurable ...	OneWay

Name	Description	Value
ServerCertificateThumbprint	Thumbprint of the server certifica...	462016f13c925aa1259a46ccbb...
TopicToPublishTo	The topic the adapter should pu...	Chamber Of Commerce
WcfBaseAddresses	Comma seperated string of Uri's	net.tcp://localhost:10045
WcfServiceName	The name of the main service.	CommunicationCenter
WcfMessageBoxServiceName	The name of the message box s...	ExternalMessageMessageBoxSe...
IsDownForMaintenance	Indicates if the adapter is down f...	False

Figure 8.4: Running Adapter Instances

8.2 Workflows

Figure 8.5 depicts the interface that lists the workflow that we have developed and placed in the runtime engines' workflow folder. We have registered the workflow by specifying a description and then pressing the Register Workflow button.



	Name	Is Two-Way
▶	Chamber of Commerce Workflow	True

Description: Chamber of Commerce Routing Workflow

Figure 8.5: Loaded Workflows

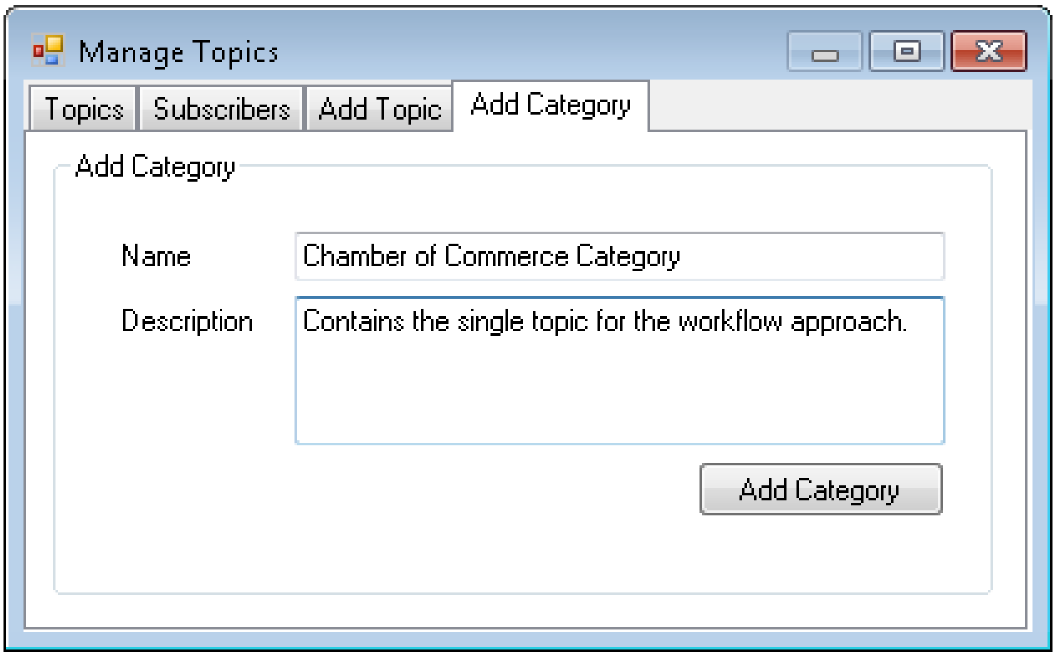
The values in the Is Two-Way column indicate if a workflow is a one-way or a two-way subscriber. This is automatically determined by analyzing the input parameters of the workflow. A workflow with a single input parameter of the internal message type with the In/Out direction it is a two-way subscriber, while if the parameter has In direction the workflow is a one-way subscriber.

Workflows that have a different direction or the wrong input type are ignored by the application and not shown in the list.

Once a workflow is registered it can be used as a subscriber of a topic.

8.3 Topics

Once the adapters and the workflow are configured we have to create a topic for them. To do this we first need to create a category to which the topic belongs using the interface depicted in Figure 8.6.

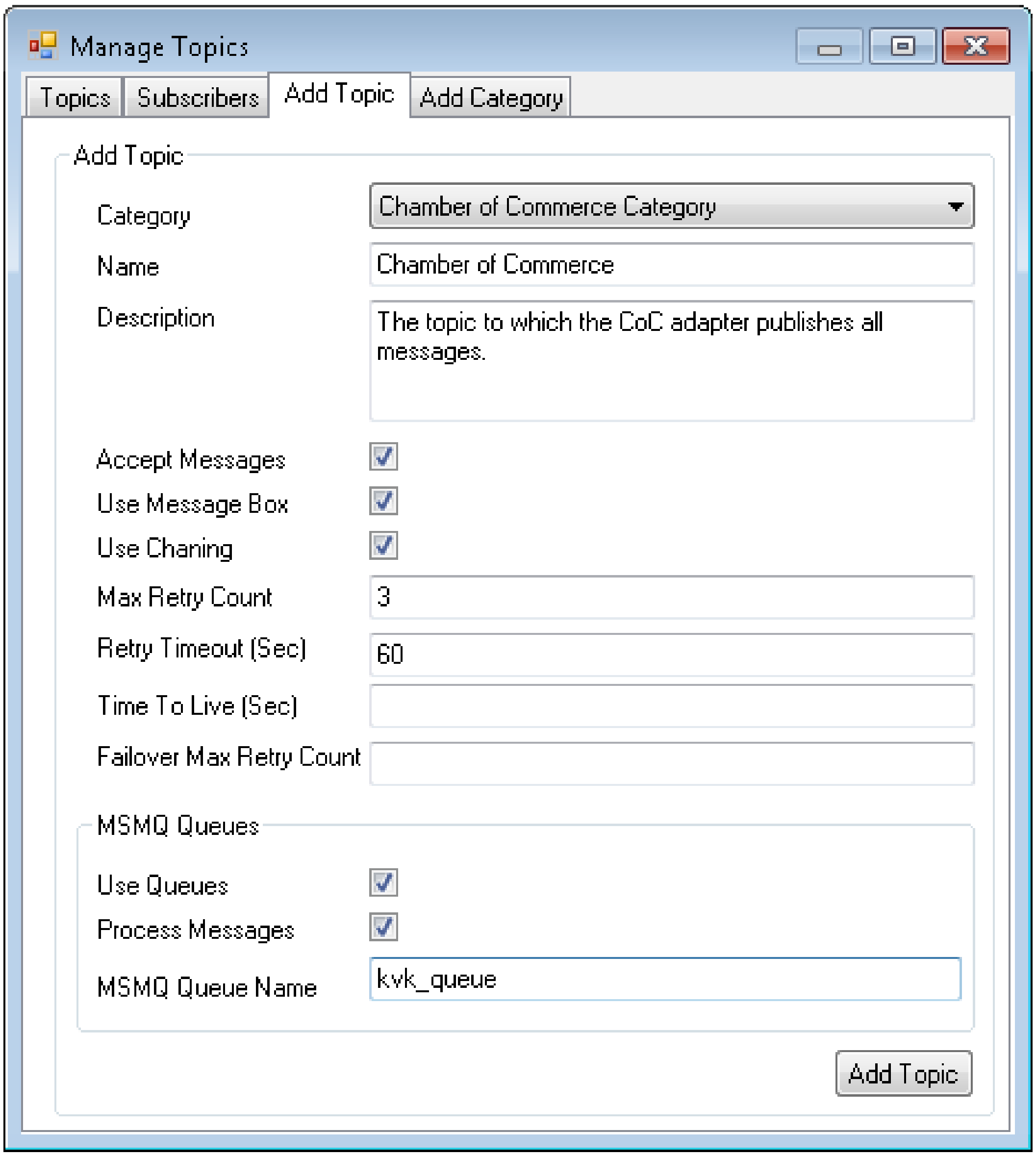


Name: Chamber of Commerce Category

Description: Contains the single topic for the workflow approach.

Figure 8.6: Category Form

Once a category has been added the interface depicted in Figure 8.7, it can be used to add topics.



Add Topic

Category	Chamber of Commerce Category
Name	Chamber of Commerce
Description	The topic to which the CoC adapter publishes all messages.
Accept Messages	✓
Use Message Box	✓
Use Chaning	✓
Max Retry Count	3
Retry Timeout (Sec)	60
Time To Live (Sec)	
Failover Max Retry Count	

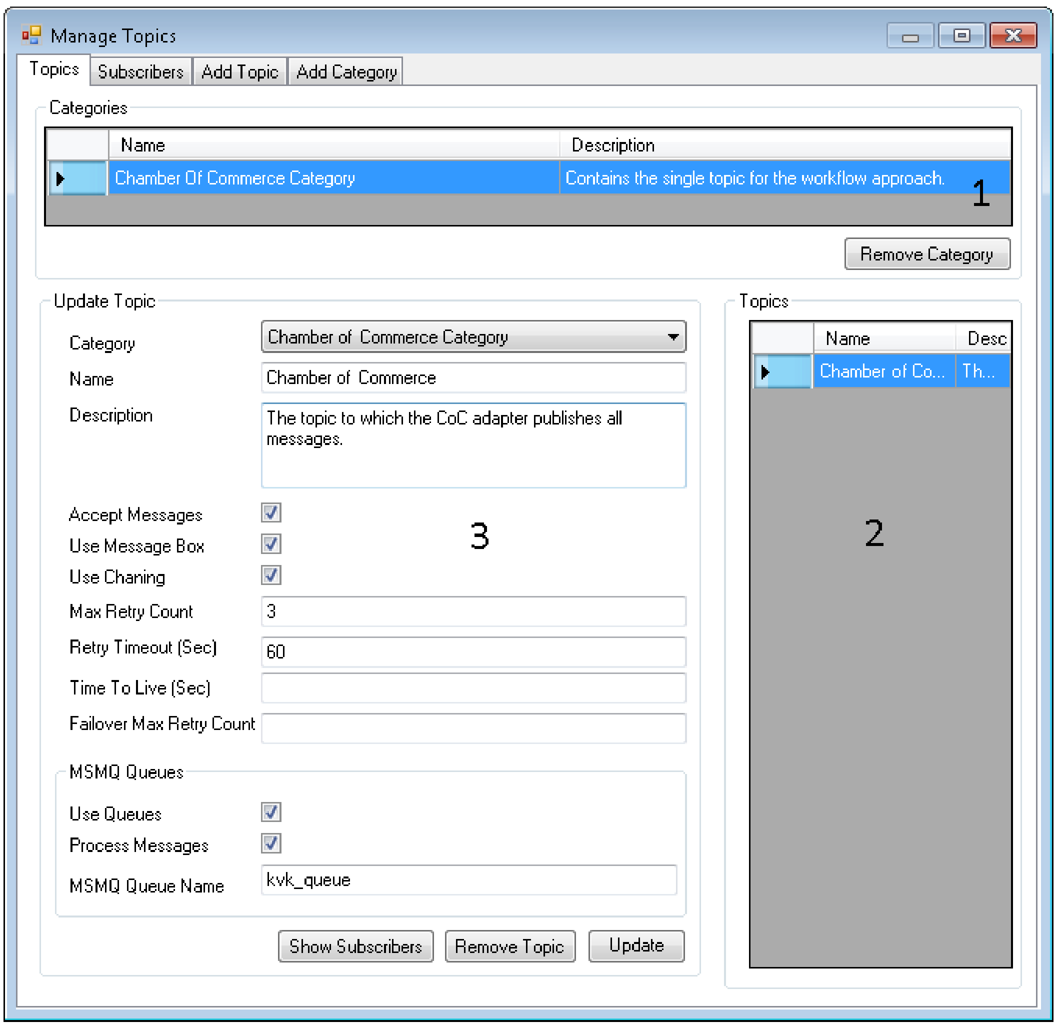
MSMQ Queues

Use Queues	✓
Process Messages	✓
MSMQ Queue Name	kvk_queue

Figure 8.7: Topics Form

We have configured the topic according to the discussed settings, so it uses the queing, chaining and message box extensions. For the MSMQ Queue Name property, we have specified a simple name of a queue that is automatically created on the server that executes the runtime engine. It is also possible to specify the path of a remote queue that runs on another server.

Figure 8.8 depicts the interface that can be used to manage existing topics.



Categories

Name	Description
Chamber Of Commerce Category	Contains the single topic for the workflow approach.

Update Topic

Category	Chamber of Commerce Category
Name	Chamber of Commerce
Description	The topic to which the CoC adapter publishes all messages.
Accept Messages	✓
Use Message Box	✓
Use Chaning	✓
Max Retry Count	3
Retry Timeout (Sec)	60
Time To Live (Sec)	
Failover Max Retry Count	

MSMQ Queues

Use Queues	✓
Process Messages	✓
MSMQ Queue Name	kvk_queue

Topics

Name	Desc
Chamber of Co...	Th...

Figure 8.8: Topics Form

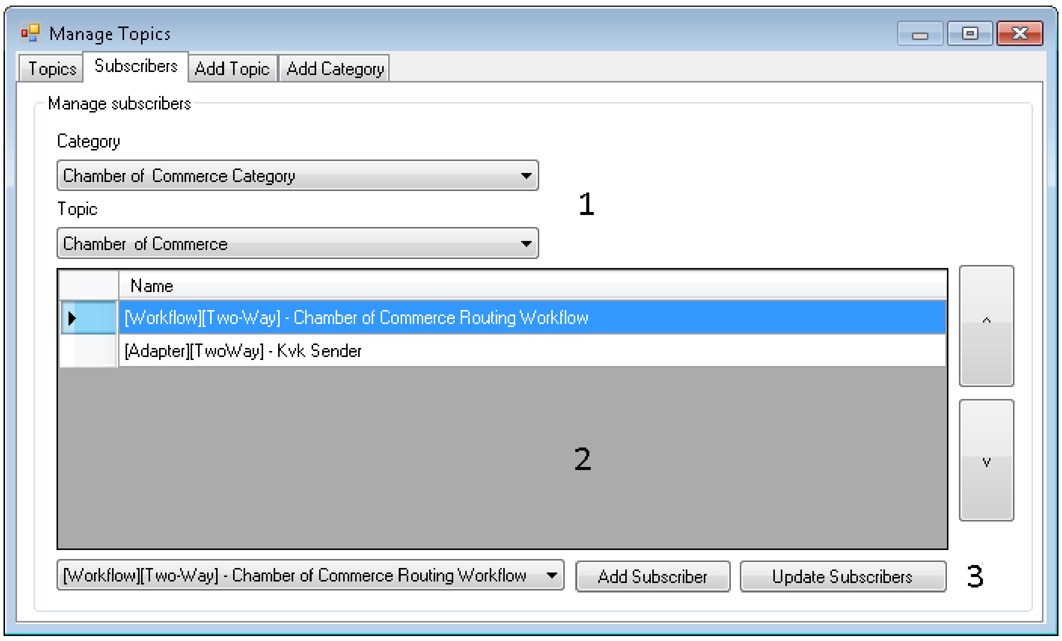
This interface has the following panes:

1. **Categories.** Can be used to select a category, and once a category is selected the topics pane (indicated with 2) displays all the topics of the selected category.

2. **Topics.** Can be used to select a topic that belongs to the selected category, and once a topic is selected the update topic pane (indicated with 3) displays the properties of the topic.

3. **Topic Properties.** Can be used to configure the topic's properties or remove the topic.

Figure 8.9 displays the interface that can be used to configure the subscribers of a topic. This interface can be shown by pressing the Show Subscribers button of the selected topic, or by pressing the Subscribers tab page.


Manage Topics

Topics | Subscribers | Add Topic | Add Category

Manage subscribers

Category: Chamber of Commerce Category

Topic: Chamber of Commerce

	Name
▶	[Workflow][Two-Way] - Chamber of Commerce Routing Workflow
	[Adapter][TwoWay] - Kvk Sender

[Workflow][Two-Way] - Chamber of Commerce Routing Workflow

Figure 8.9: Topics Subscribers

This interface has the following panes:

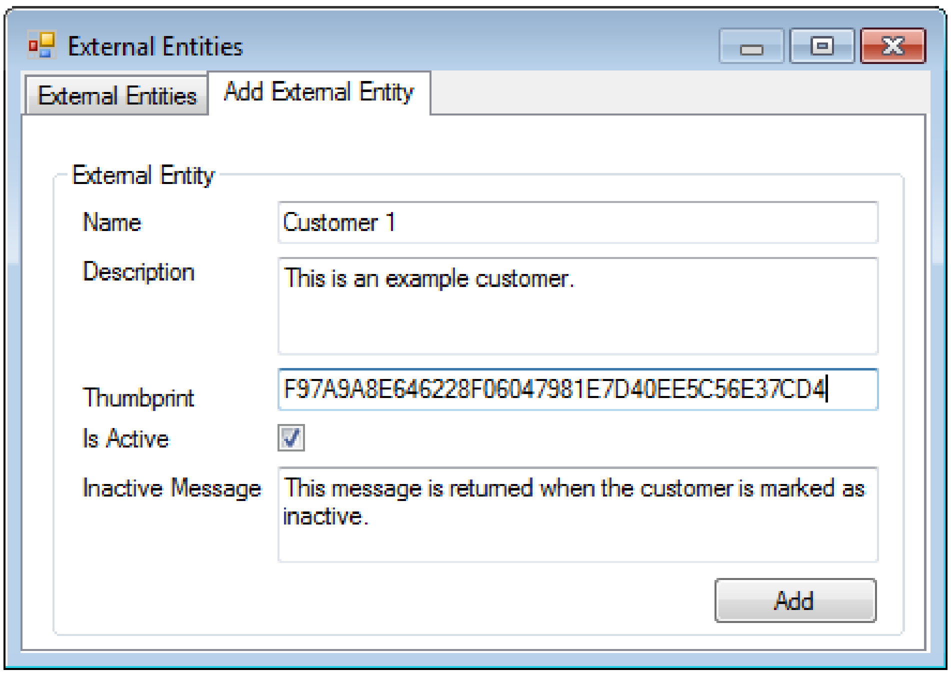
1. **Topic Selection.** First a manager has to select the intended topic. If the Show Subscribers button depicted in Figure 8.8 was pressed the topic is already selected, otherwise this can be done by using the combo boxes to select the proper category and topic.

2. **Ordered Subscriber List.** This control lists the subscribers of the selected topic. The order of the subscribers can be changed with the up and down buttons on the right, and subscribers can be deleted by pressing the delete button on the keyboard.

3. **Updating Subscribers.** The combo box on the bottom left corner of Figure 8.9 displays all adapter instances that do not have the None MEP, and it also displays all the registered workflows. The Add Subscriber button can be used to add the selected subscriber to the list. Finally, the Update Subscriber button can be pressed to send the updated list of subscribers to the runtime engine.

8.4 External Entities

Finally the external entities that are allowed to communicate with the runtime engine should be added using the interface depicted in Figure 8.10.



Name: Customer 1

Description: This is an example customer.

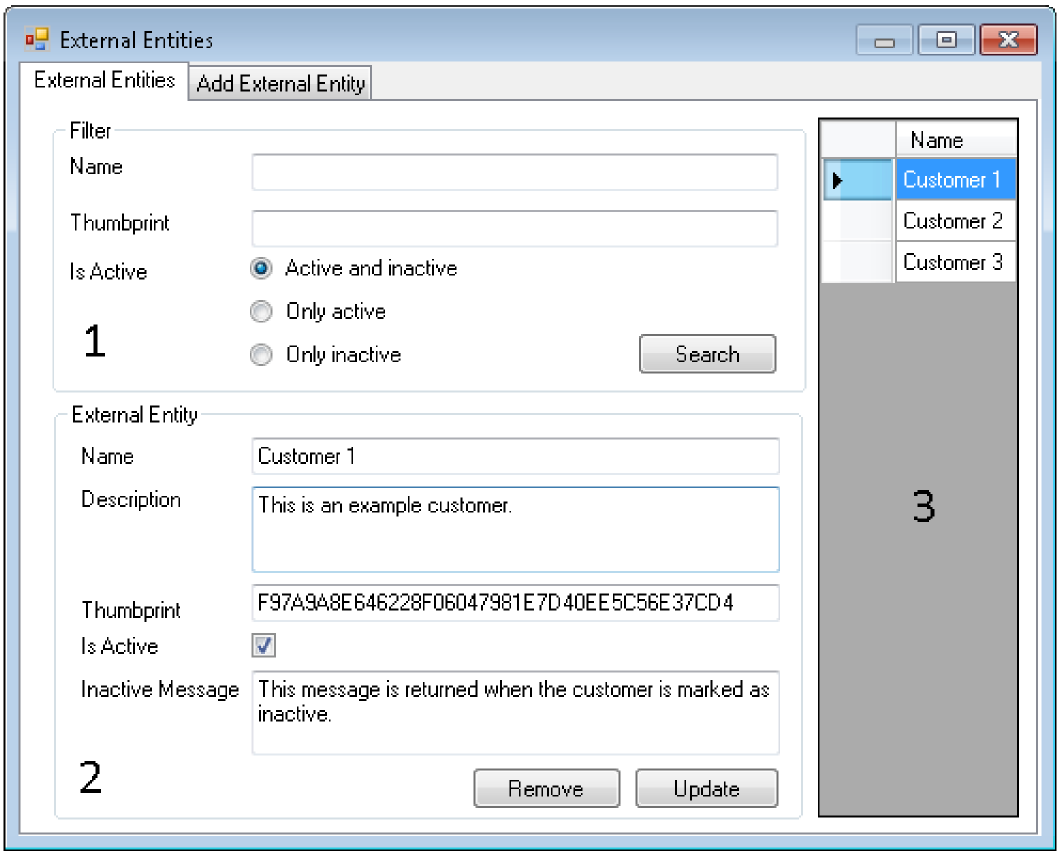
Thumbprint: F97A9A8E646228F06047981E7D40EE5C56E37CD4

Is Active: ✓

Inactive Message: This message is returned when the customer is marked as inactive.

Figure 8.10: Add External Entity Interface

The external entities added before can be managed with the interface depicted in Figure 8.11.



Filter

Name:

Thumbprint:

Is Active: [✓ Active and inactive] Only active Only inactive

External Entity

Name: Customer 1

Description: This is an example customer.

Thumbprint: F97A9A8E646228F06047981E7D40EE5C56E37CD4

Is Active: ✓

Inactive Message: This message is returned when the customer is marked as inactive.

	Name
▶	Customer 1
	Customer 2
	Customer 3

Figure 8.11: Manage External Entity Interface

This interface has the following panes:

1. **Filter Criteria.** Provides the ability to specify filter criteria to display specific external entities.
2. **Selected External Entity.** Displays and provides the ability to update or remove a selected external entity.
3. **External Entity Grid.** Displays the external entities that match the specified filter criteria.

The Is Active checkbox allows managers to temporarily disable external entities without having to complectly remove them. If an external entity is set to inactive, the publish-subscribe engine rejects all its messages, and immediately returns a publish result with (1) an error indicating that the external entity is inactive and (2) the specified Inactive Message.

8.5 Logged Messages

Once the runtime engine is configured, the management application can be used to view the complete flow of messages through the runtime engine. The runtime engine logs messages using the model that was discussed in Section 5.2, which provides the three log messages depicted in Figure 8.12 for each message that is received from a CoC connector.

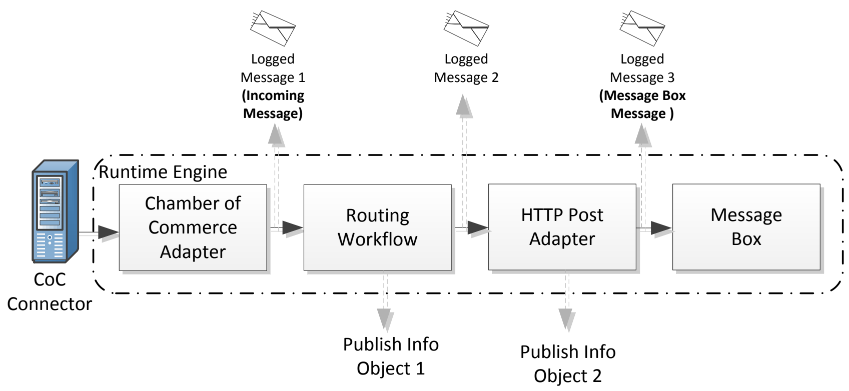


Figure 8.12: Log Locations

1. **Logged Message 1.** The first logged message is called an incoming message and is logged when the CoC adapter publishes an internal message to the publish-subscribe engine.

   Incoming messages are the original messages published to a topic, and not the replies received from subscribers of a topic.

2. **Logged Message 2.** The second message is logged when the routing workflow returns the enhanced message.

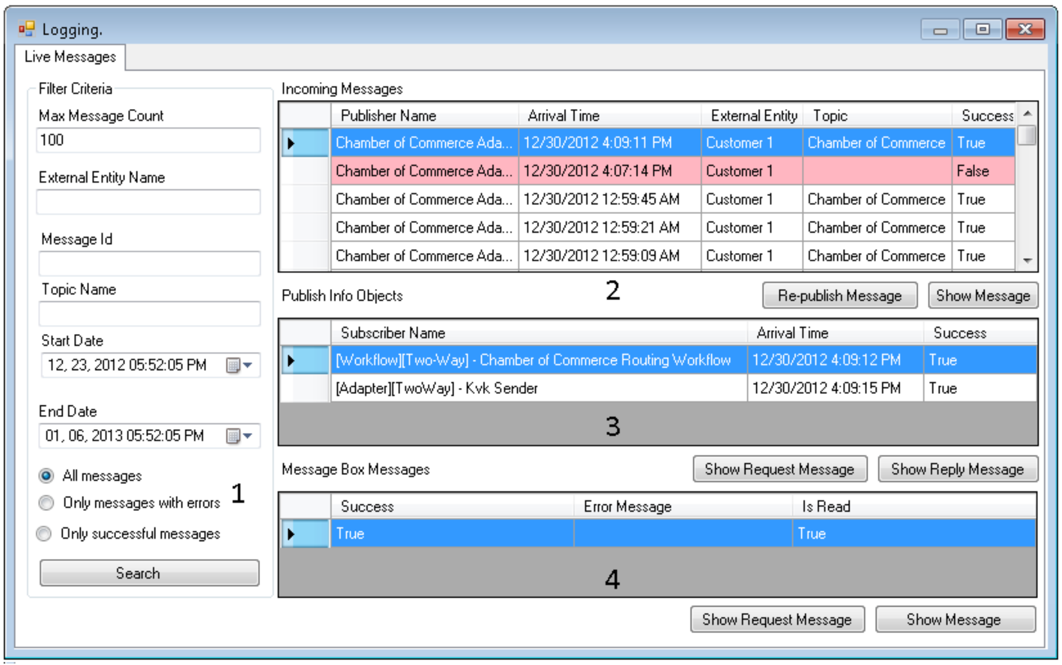
3. **Logged Message 3.** The last message is called a message box message and it is logged when the HTTP Post adapter returns the result from the CoC service.

   Message box messages are stored in the message box to make it possible to retrieve them asynchronously.

Using the workflow approach the runtime engine logs two publish info objects and for each publish info object the request message and the response message is logged. For the first publish info object the request message is message 1, and the reply message is message 2.

If a subscriber throws an error, the publish-subscribe engine retries to publish the message to the subscriber, and for each try, a new reply message and a new publish info object is logged and displayed.

The logged messages and logged publish info objects can be viewed by using the interface depicted in Figure 8.13:



Filter Criteria

Max Message Count	100
External Entity Name	
Message Id	
Topic Name	
Start Date	12, 23, 2012 05:52:05 PM
End Date	01, 06, 2013 05:52:05 PM

[✓ All messages] Only messages with errors Only successful messages

Incoming Messages

Publisher Name	Arrival Time	External Entity	Topic	Success
Chamber of Commerce Ada...	12/30/2012 4:09:11 PM	Customer 1	Chamber of Commerce	True
Chamber of Commerce Ada...	12/30/2012 4:07:14 PM	Customer 1		False
Chamber of Commerce Ada...	12/30/2012 12:59:45 AM	Customer 1	Chamber of Commerce	True
Chamber of Commerce Ada...	12/30/2012 12:59:21 AM	Customer 1	Chamber of Commerce	True
Chamber of Commerce Ada...	12/30/2012 12:59:09 AM	Customer 1	Chamber of Commerce	True

Publish Info Objects

Subscriber Name	Arrival Time	Success
[Workflow][Two-Way] - Chamber of Commerce Routing Workflow	12/30/2012 4:09:12 PM	True
[Adapter][TwoWay] - Kvk Sender	12/30/2012 4:09:15 PM	True

Message Box Messages

Success	Error Message	Is Read
True		True

Figure 8.13: Logging Interface

This interface has the following panes:

1. **Filter Criteria.** Provides the ability to specify filter criteria to display specific logged messages.

2. **Incoming Messages.** Displays the messages that match the specified filter criteria. This pane only displays incoming messages, unless the filter criteria specifies a specific id of another message. Unsuccessfully published incoming messages are marked red.

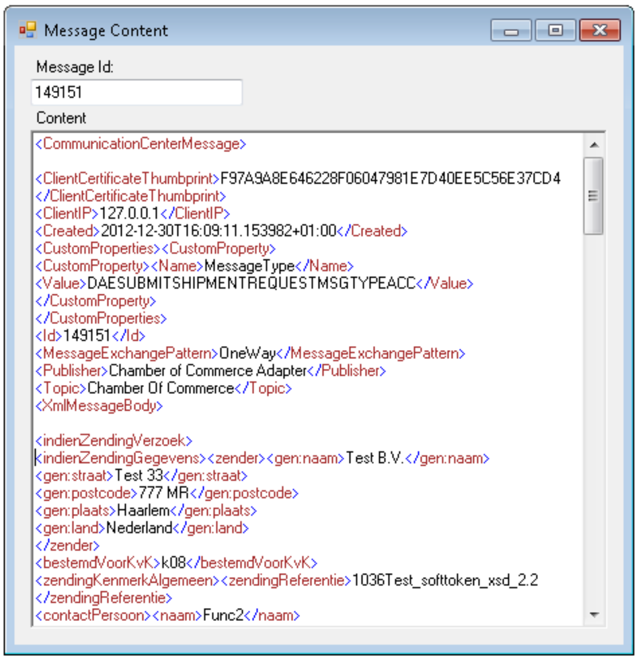
3. **Publish Info Objects.** Displays the publish info objects that are logged for the selected incoming message.

4. **Message Box Messages.** Displays the message box messages that are logged for the selected incoming internal message.

8.5.1 Message Content

The same interface can be used to view the complete internal messages in XML format by pressing an incoming message and clicking the Show Message button, or by selecting a publish info object, or message box message and pressing the Show Request Message or Show Response Message buttons.

Figure 8.14 depicts an incoming message in XML format that allows managers to see the metadata and the actual payload of an internal message.

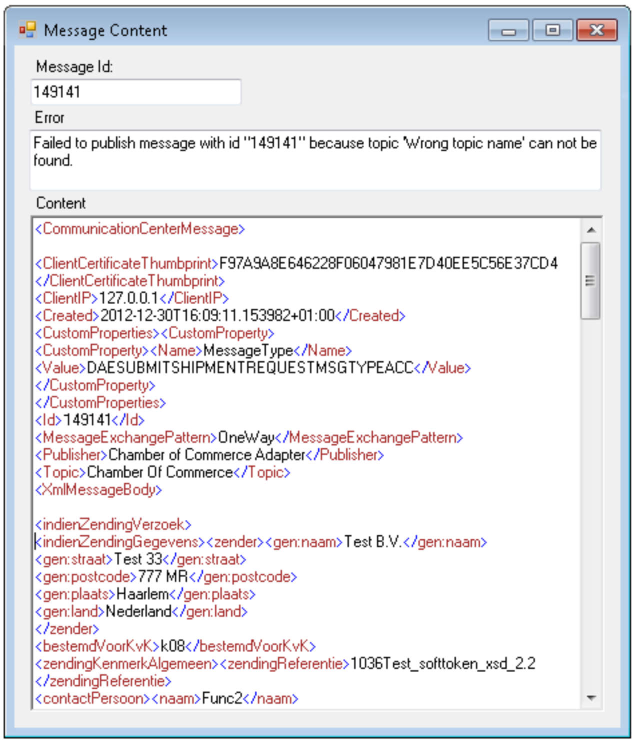


```
<CommunicationCenterMessage>

<ClientCertificateThumbprint>F97A9A8E646228F06047981E7D40EE5C56E37CD4
</ClientCertificateThumbprint>
<ClientIP>127.0.0.1</ClientIP>
<Created>2012-12-30T16:09:11.153982+01:00</Created>
<CustomProperties><CustomProperty>
<CustomProperty><Name>MessageType</Name>
<Value>DAESUBMITSHIPMENTREQUESTMSGTYPEACC</Value>
</CustomProperty>
</CustomProperties>
<Id>149151</Id>
<MessageExchangePattern>OneWay</MessageExchangePattern>
<Publisher>Chamber of Commerce Adapter</Publisher>
<Topic>Chamber Of Commerce</Topic>
<XmlMessageBody>

<indienZendingVerzoek>
<indienZendingGegevens><zender><gen:naam>Test B.V.</gen:naam>
<gen:straat>Test 33</gen:straat>
<gen:postcode>777 MR</gen:postcode>
<gen:plaats>Haarlem</gen:plaats>
<gen:land>Nederland</gen:land>
</zender>
<bestemdVoorKvK>k08</bestemdVoorKvK>
<zendingKenmerkAlgemeen><zendingReferentie>1036Test_softtoken_xsd_2.2
</zendingReferentie>
<contactPersoon><naam>Func2</naam>
```

Figure 8.14: Incoming Message From The CoC Connector

If a message contains an error, like the second incoming message in Figure 8.13 the Show Message buttons can be used to visualize the error.

Figure 8.15 shows the the interface that displays an internal message that was rejected by the publish-subscribe engine because an invalid topic was specified.



Message Id:
149141
Error
Failed to publish message with id "149141" because topic 'Wrong topic name' can not be found.
Content

```
<CommunicationCenterMessage>

<ClientCertificateThumbprint>F97A9A8E646228F06047981E7D40EE5C56E37CD4
</ClientCertificateThumbprint>
<ClientIP>127.0.0.1</ClientIP>
<Created>2012-12-30T16:09:11.153982+01:00</Created>
<CustomProperties><CustomProperty>
<CustomProperty><Name>MessageType</Name>
<Value>DAESUBMITSHIPMENTREQUESTMSGTYPEACC</Value>
</CustomProperty>
</CustomProperties>
<Id>149141</Id>
<MessageExchangePattern>OneWay</MessageExchangePattern>
<Publisher>Chamber of Commerce Adapter</Publisher>
<Topic>Chamber Of Commerce</Topic>
<XmlMessageBody>

<indienZendingVerzoek>
<indienZendingGegevens><zender><gen:naam>Test B.V.</gen:naam>
<gen:straat>Test 33</gen:straat>
<gen:postcode>777 MR</gen:postcode>
<gen:plaats>Haarlem</gen:plaats>
<gen:land>Nederland</gen:land>
</zender>
<bestemdVoorKvK>k08</bestemdVoorKvK>
<zendingKenmerkAlgemeen><zendingReferentie>1036Test_softtoken_xsd_2.2
</zendingReferentie>
<contactPersoon><naam>Func2</naam>
```

Figure 8.15: Unsuccessfully Published Message

8.6 Adapter Approach Configuration

The management applications can also be used to configure the runtime engine according to the adapter approach using the interfaces discussed before. To do this, the following steps have to be performed:

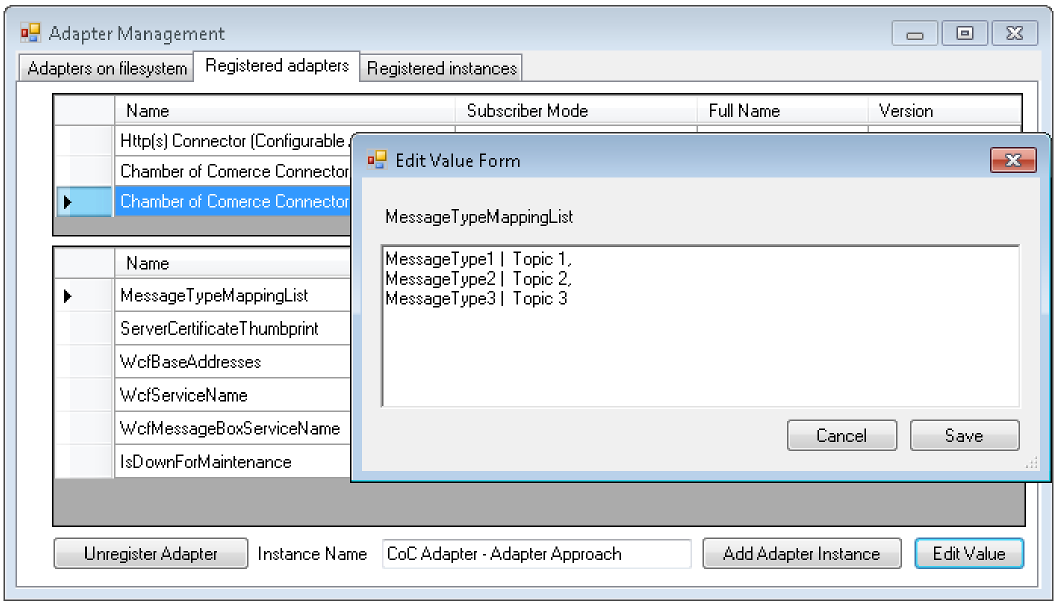
1. Registering and instantiating the variation of the CoC connector that declares a property, which allows managers to specify the mapping between the message type of a message and the topic the message has to be published to.

   Figure 8.16 depicts the interface that can be used to enter list values such as the message type to topic mapping.

2. For each message type configured in the mapping a topic has to be created.

3. Each topic must have an instance of the HTTP Post adapter that is configured with the endpoint address of the intended CoC service.

The other steps are similar to the ones discussed for the configuration of the runtime engine for the workflow approach and shown in Figures 8.2 to 8.15.



Name	Subscriber Mode	Full Name	Version
Http(s) Connector (Configurable			
Chamber of Comerce Connector			
Chamber of Comerce Connector			

Name
MessageTypeMappingList
ServerCertificateThumbprint
WcfBaseAddresses
WcfServiceName
WcfMessageBoxServiceName
IsDownForMaintenance

MessageTypeMappingList

```
MessageType1 |  Topic 1,
MessageType2 |  Topic 2,
MessageType3 |  Topic 3
```

Instance Name: CoC Adapter - Adapter Approach

Figure 8.16: Register Adapter Approach CoC Adapter

Chapter 9

Architecture Evaluation

This chapter evaluates the architecture of the PCC2 through the prototype we discussed in Chapter 6, and is structured as follows:
Section 9.1 evaluates the case study.
Section 9.2 checks whether the prototype fulfils the requirements.

9.1 Case Evaluation

In the case study we used the prototype to provide two solutions for the CoC integration scenario. The PCC also provides a solution for the CoC integration scenario, which works as described in Appendix A.2. We compared the PCC2 with the PCC by analyzing their ability to apply changes to their solutions without downtime.

This is done by using the evaluation criteria that we have defined based on analysis of the CoC integration scenario, and from the experience of the PCC designers.

Below the specified evaluation criteria are listed:

C1: Is it possible to modify the endpoint addresses of the CoC services without downtime?
This is required in case the CoC updates its services and assigns new addresses to them.

C2: Is it possible to support a new message type that needs to be sent to a new endpoint address without downtime?
This is needed when the CoC provides support for a new export document that must be made available to the customers.

C3: Is it possible to modify the XML message format of the messages that need to be sent to the CoC services without downtime?
If the system is able to transform messages it is possible to transition between different versions without immediately requiring a new version of all clients. This can

be used when the CoC modifies an existing export document and releases a new version of one of its XSD schemas.

C4: Does the system prevent message loss if the system is unexpectedly restarted?
Message loss has to be prevented because customers pay for each message they send, so it is not acceptable to lose messages if the system is unexpectedly restarted.

C5: Are reply messages still delivered if an external entity is unavailable for several hours?
Message loss also has to be prevented when a customer sends a message, and then encounters technical difficulties that prevent the customer from retrieving the reply message for several hours.

C6: Is it possible to accept a new client that sends messages in a different message format without downtime?
This is needed because in the future new clients that possibly use unsupported message formats, have to be able to communicate with the system.

C7: How much time does it take to process 1000 messages and receive the replies?
This evaluation criterion provides an indication of the processing times of the PCC and PCC2.

Section uses evaluation criteria C1 to C6 to evaluate the PCC, and Section Section 9.1.2 does the same for the PCC2. We acquired the PCC evaluation results by interviewing the PCC developers, and used our own knowledge to provide the PCC2 evaluation results.

Evaluation criteria C7 is used to evaluate the performance of the PCC and the PCC2, and is covered in Section 9.1.3.

9.1.1 PCC Evaluation

This section evaluates the PCC using the specified evaluation criteria.

C1: Is it possible to modify the endpoint addresses of the CoC services without downtime?
Yes, but they have to be modified directly in the database.

C2: Is it possible to support a new message type that needs to be sent to a new endpoint address without downtime?
No, this is not possible without any downtime because the PCC Message Processing program has to be updated, and for that to happen it has to be stopped, the code has to be modified and finally the program has to be started again.

C3: Is it possible to modify the XML message format of the messages that need to be sent to the CoC services without downtime?
No, this also requires the Message Processing program to be updated.

C4: Does the system prevent message loss if it is unexpectedly restarted?
Yes, because the messages are stored in queues that are used to recover messages in case of an unexpected restart.

C5: Are reply messages still delivered if an external entity is unavailable for several hours?
No, if an external entity is unavailable the PCC tries three times to send the reply messages to the external entity, but if all tries fail the messages are stored in the dead-letter queue and are not sent again.

C6: Is it possible to accept a new client that sends messages in a different message format without downtime?
No, this also requires the Message Processing program to be updated which causes downtime.

9.1.2 PCC2 Evaluation

This section evaluates the PCC2 using the specified evaluation criteria.

C1: Is it possible to modify the endpoint addresses of the CoC services without downtime?
Yes, this is possible using two approaches:

1. If the workflow approach is used, the routing workflow can be modified, and once it is saved the new addresses can be used.

2. If the adapter approach is used, the properties of the HTTP Post adapter instances can be updated using the management application.

C2: Is it possible to support a new message type that needs to be sent to a new endpoint address without downtime?
Yes, this is possible using two approaches:

1. If the workflow approach is used, the routing workflow has to be be modified to include the new message type and endpoint address. Once the workflow is modified and saved the new message type is accepted.

2. If the adapter approach is used, the following steps have to be taken to accept the new message type:

    - A new topic has to be created for the new message type.
    - The created topic requires an instance of the HTTP Post adapter that is configured with the correct endpoint address as subscriber.
    - The CoC adapter's message type mapping has to be updated to include the created topic.

C3: Is it possible to modify the XML message format of the messages that need to be sent to the CoC services without downtime?
Yes, this is possible using two approaches:

1. If the workflow approach is used, the existing workflow can be modified to perform the transformations, or a second workflow can be developed and deployed to perform the transformations.

2. In the adapter approach, a new workflow has to be developed and deployed to perform the transformations.

C4: Does the system prevent message loss if it is unexpectedly restarted?
Yes, because the messages are stored in queues that are used to recover messages in case of an unexpected restart.

C5: Are reply messages still delivered if an external entity is unavailable for several hours?
Yes, they are stored in the message box until an external entity retrieves them.

C6: Is it possible to accept a new client that sends messages in a different message format without downtime?
Yes, this is possible (using both approaches) by developing a new adapter that can communicate with the client, and developing a workflow that transforms the messages to the correct format. The management application can then be used to configure the developed adapter and workflow without downtime of the already operational adapters and workflows.

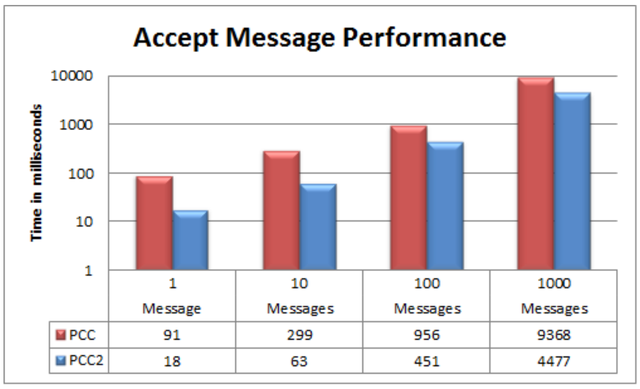
9.1.3 Performance Evaluation

This section compares the performance of the PCC and the PCC2, and covers evaluation criteria C7, by using a test client that we developed to send a configurable amount of messages to the PCC and the PCC2, and measure the time it takes to accept and process them.

The test setup is as follows:

- The test client, the PCC and the PCC2 all run on a machine running Windows 7, 4 GB RAM on a 2.83GHz QuadCore processor.
- The test client sends an export document in XML format of 4534 characters to the PCC and PCC2.
- The PCC2 is configured according to the workflow approach because this approach is discussed in detail in Chapter 8, and the performance of the adapter approach only differs in the order of milliseconds.
- Because the CoC test environment is slow, and can not handle high message loads we modified the PCC and PCC2 to simulate the calls to the CoC services. To do this we modified the PCC Outgoing Frontend (depicted in Figure 2.3), and the PCC2 HTTP Post Adapter (depicted in Figure 7.12) to wait one second, and then return a XML reply message of 1116 characters to indicate a submitted export document was accepted.
- All measurements are an average of 10 runs.

The first test only measures how much time it takes to accept messages, which means accept the export document in XML format via a Web service, store it in a queue and send a synchronous reply to indicate the message was successfully accepted. Figure 9.1 shows the results.



	1 Message	10 Messages	100 Messages	1000 Messages
🟥 PCC	91	299	956	9368
🟦 PCC2	18	63	451	4477

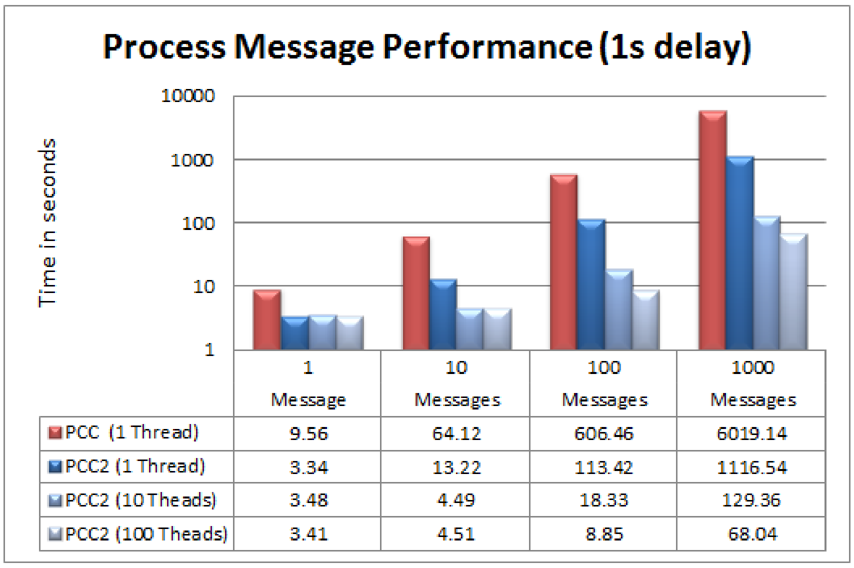
Figure 9.1: Accept Message Performance

The PCC and the PCC2 both accept messages in linear time, but the PCC2 accepts messages slightly faster than the PCC because the PCC Incoming Frontend performs a Web service call to the Incoming Message Handler to store messages in the queue, while the PCC2 does not require this additional call.

The second test measures how much time it takes to accept and process messages. We performed the PCC2 tests with 1, 10 and 100 processing threads that simultaneously process the accepted messages stored in the queue. We did this because the PCC can only processes messages with a single thread, while the PCC2 can processes them with a configurable amount of threads.

To receive the reply messages from the PCC, the test client hosts a Web service that is called by the PCC, and to receive the reply messages from the PCC2 the test client polls the HTTP Post Adapter with an interval of 1 second.

Figure 9.2 shows the results.



	1 Message	10 Messages	100 Messages	1000 Messages
🟥 PCC (1 Thread)	9.56	64.12	606.46	6019.14
🟦 PCC2 (1 Thread)	3.34	13.22	113.42	1116.54
🟦 PCC2 (10 Theads)	3.48	4.49	18.33	129.36
⬜ PCC2 (100 Theads)	3.41	4.51	8.85	68.04

Figure 9.2: Process Message Performance

The results indicate that the PCC2 processes the accepted messages faster than the PCC, even if only one processing thread is used. This is because the PCC2 centralized architecture provides less overhead, and because the PCC2 is implemented using newer and faster implementation technologies.

How much the PCC2 is faster than the PCC depends on the amount of concurrent calls a third-party service can process, the poll interval of the client, and the time it takes the third-party service to process a single message.

C7: How much time does it take to process 1000 messages and receive the replies? To answer this evaluation criteria we used the values for the PCC and PCC2 depicted in in Figure 9.2. For the PCC2 we used the value that was measured using using 10 processing threads, because that value is comparable to the time it would take to send a 1000 messages to the CoC production environment.

This is because, just like our test setup, the CoC production environment allows the PCC2 to use 10 processing threads, and also takes approximately 1 second to process a single message.

The only difference is that the poll interval of the PCC2 clients might be a couple of seconds slower, but this is negligible on the total time when sending a 1000 messages.

The results are:

- **PCC2:** 129.36s using 10 processing threads.
- **PCC:** 6019.14s using 1 processing thread.

This means that the PCC2 processes a 1000 messages approximately 46.53 times faster than the PCC, when communicating with the CoC production environment.

9.1.4 Comparison

This section provides an overview of the evaluation results and compares the PCC and the PCC2. We have used the following ratings for the comparison:

- The + rating was given if a solution met the criteria, or if a solution provided a better result than the other solution.
- The - rating was given if a solution did not meet the criteria, or if a solution provided a weaker result than the other solution.

Table 9.1 provides the overview of the evaluation results.

Table 9.1: Case study evaluation results

Evaluation Criteria	PCC	PCC2
C1: Is it possible to modify the endpoint addresses of the CoC services without downtime?	+ (Yes)	+ (Yes)
C2: Is it possible to support a new message type that needs to be sent to a new endpoint address without downtime?	− (No)	+ (Yes)
C3: Is it possible to modify the XML message format of the messages that need to be sent to the CoC services without downtime?	− (No)	+ (Yes)
C4: Does the system prevent message loss if it is unexpectedly restarted?	+ (Yes)	+ (Yes)
C5: Are reply messages still delivered if an external entity is unavailable for several hours?	− (No)	+ (Yes)
C6: Is it possible to accept a new client that sends messages in a different message format without downtime?	− (No)	+ (Yes)
C7: How much time does it take to process 1000 messages and receive the replies?	− (1.67h)	+ (2.16m)

Based on the results we can see that the PCC2 is faster and more flexible to apply changes to its solutions without downtime than the PCC.

9.1.5 Case Conclusions

The case study in Chapter 7 provides a detailed explanation of the PCC2 capabilities by discussing all steps that have to be performed to develop and deploy a solution for an integration scenario.

It also demonstrates that the PCC2 supports different approaches (adapter and workflow-based) to provide a solution for the same integration scenario, and we also compared these approaches.

An advantage of the adapter approach is that only the management application is needed for the complete configuration, and it does not require an external application like Visual Studio to modify the workflow as in the other approach.

An advantage of the workflow approach is that the configuration of the management application is simple compared to the configuration of the adapter approach. This is because only a single topic needs to be created and configured instead of defining a topic for each message type. However, depending on the configuration requirements of a specific application this can also be seen as a disadvantage because it provides less possibilities.

Using the adapter approach, each message of a different message type is published to a different topic, which makes it possible to provide a different configuration per topic. This allows managers to, for example, temporarily disable the processing of a single message type, or specify different retry options per message type.

Whatever approach is chosen, the PCC2 is more efficient and provides developers the flexibility to modify its solutions to cope with changes that might occur. This is because it encapsulated the integration scenario specific source code in reusable adapters and workflows that can be flexibly added and updated at runtime without requiring modification to the PCC2 source code.

9.2 Requirements Evaluation

This section evaluates the prototype based on the requirements that we defined in Chapter 3.

By using the requirements as criteria for the evaluation we can on the one hand evaluate the prototype against useful criteria and on the other hand check whether the requirements of the architecture are met.

9.2.1 Development Level Requirements

This section describes how the Development level requirements are fulfilled.

R1: The system has to provide a mechanism to host and communicate with additional endpoints without disrupting the already running applications.
The runtime engine provides an adapter framework that allows developers to develop and deploy reusable adapters that can communicate with an additional endpoint that each can support a different MEP, communication protocol and message format without disrupting the already running adapters.

The adapter framework also makes it possible to host new endpoints, which can be done, for example, by developing an adapter that hosts a Web service using the Service Host Factory, and instantiating multiple instances of this adapter using the management application. Each instance then hosts a new endpoint that clients can use to communicate with the runtime engine.
▹ *Section 7.4* provides an explanation of the HTTP Post Adapter, which is an example of how the adapter framework can be used to communicate with additional endpoints without disrupting the already running applications.
▹ *Section 7.3* provides an explanation of the CoC adapter, which is an example of how the adapter framework can be used to host new endpoints.

R2: The system has to provide a flexible mechanism to perform message processing.
The workflow framework provides a flexible way to develop and deploy reusable WF4.5 workflows, which can be used to implement custom business logic or perform message processing.

These workflows can leverage the full .NET Framework, which allows them to:

- Performing XSD-Schema validations.

- Call Web services or databases to find additional information to be add to the message.
- Apply XSLT transformations to transform the messages.

The workflows are fully updatable at runtime without affecting already running adapters and workflows.
▹ *Section 6.6* provides an explanation of an example message processing workflow.

9.2.2 Operational Level Requirements

This section describes how the Operational level requirements are fulfilled.

R3: The system has to provide a mechanism to support asynchronous communication without external entities having to open inbound ports in their firewalls.
The message box extension allows managers to configure the publish-subscribe engine to store the processing results of a message in the message box, which allows adapter developers to asynchronously make these results available to external entities without requiring them to open any inbound ports in their firewalls.

The processing results can be the reply message of a subscriber or an error message indicating that processing errors occurred.
▹ *Section 5.6* describes the message box extension.
▹ *Section 7.3* provides an explanation of the CoC adapter, that uses the message box extension.

9.2.3 Management Level Requirements

This section describes how the Management level requirements are fulfilled.

R4: The system has to provide a flexible mechanism to perform message routing.
The publish-subscribe engine allows managers to perform itinerary-based routing by using the management application to specify a message itinerary per topic that is used to publish messages to subscribers.

It is also possible to perform content-based routing by using properties of an adapter or by using a routing workflow that can be deployed in the workflow framework.

As the case study has shown, both approaches are fully updatable at runtime without affecting already running adapters and workflows.
▹ *Chapter 7* demonstrates how the PCC2 supports both routing mechanisms.

R5: The system must only accept incoming messages from predefined external entities.
The publish-subscribe engine only accepts messages from external entities that have a valid X.509 certificate and have been registered using the management application.
▹ *Section 8.4* explains how the management application can be used to manage known external entities.

R6: The system has to be able to temporarily reject incoming messages.
A manager can use the management application to set the AcceptMessages property of a topic to false, which causes the publish-subscribe engine to reject all messages that are published to that topic.
▹ *Section 5.1* describes the AcceptMessages property.
▹ *Section 8.7* depicts the interface of the management application that can be used to set the AcceptMessages property.

R7: The system has to provide a configurable retry mechanism.
The publish-subscribe engine provides a configurable retry mechanism that can be configured using the management application.
▹ *Section 5.3* describes the configurable retry mechanism.
▹ *Section 8.7* depicts the interface of the management application that can be used to configure the retry mechanism.

R8: The system has to be able to display detailed log information.
The runtime engine provides detailed log information about all sent and received messages that allows managers to use the management application to view the complete flow of messages through the runtime engine.

This view includes:

- The name of the external entity that sent a message.
- At what time a message was sent to a recipient and the reply that was received for the message.

- Which errors, if any, occurred during the processing of a message.
- The final reply message that was generated by the system.

▹ *Section 5.2* describes the the model that is used to provide the detailed log information.
▹ *Section 8.5* describes the interface that can be used to view the detailed log information.

OR10: The system has to provide support for a fallback address for external entities.
The retry mechanism of the publish-subscribe engine has the ability to publish a message in fail over mode, which sets the UseFailover metadata property of the internal message to true before a failed message is re-published.

The subscribers can then take appropriate actions, such as, deliver the message to a fail over endpoint.
▹ *Section 5.3* describes how this fallback mechanism can be used.
▹ *Section 8.7* depicts the interface of the management application that can be used to enable this mechanism.

OR15: The system has to collect usage information.
Managers can view usage information of external entities by using the management application to filter all logged messages on the name of a specific external entity.
▹ *Section 5.2* describes the the model that is used to provide the usage information.
▹ *Section 8.5* describes the interface that can be used to view and filter the logged messages to view the usage information.

OR16: The system must be able to temporarily stop processing messages.
The ProcessMessages property of a topic allows managers to temporarily disable message processing of a specific topic. This property indicates if the publish-subscribe engine has to process messages that have been stored in the queue. If this option is set to false, the publish-subscribe engine still accepts incoming messages and stores them in the queue, but they are not processed until this option is enabled.
▹ *Section 5.1* describes the ProcessMessages property.
▹ *Section 8.7* depicts the interface of the management application that can be used to set the ProcessMessages property.

OR17: The system must be able to temporarily block external entities.

The IsActive property of an external entity allows managers to temporarily disable external entities without having to complectly remove them. If an external entity is set to inactive, the publish-subscribe engine rejects all its messages and immediately returns a publish result that includes: (1) an error indicating that the external entity is inactive, (2) an description that was specified by a manager.

▹ *Section 8.4* depicts the interface of the management application that can be used to block external entities.

9.2.4 Requirements Conclusions

The requirements evaluation results show that the requirements of the project were met because the PCC2 fulfils all mandatory and four optional requirements.

Based on the results we concluded that the PCC2 satisfies the needs of the three identified stakeholder groups, i.e. managers, developers and external as follows:

- **Managers.**

  The PCC2 provides a management application that increases the productivity of the managers by allowing multiple managers to simultaneously configure the PCC2 from a remote location without having to redeploy the PCC2. Troubleshooting problems also takes a lot less time because the management application provides detailed log information about all sent and received messages.

- **Developers**

  The PCC2 provides the developers with a Service Host Factory, and a workflow and adapter framework, that increase their productivity by allowing them to easily develop and deploy reusable components to communicate with additional endpoints without downtime. In addition, the PCC2 configurable publish-subscribe engine decouples the developed components, allowing them to effectively exchange messages by simply publishing messages onto the bus, independent of the type or number of consumers. This also increases the developers' productivity because it frees them from spending time on defining the communication between the developed components, and allows them to concentrate on the specific business logic associated with manipulating the message content.

- **External entities** The PCC2 allows external entities to easily communicate with the PCC2 without having to modify their system by allowing the developers to easily develop components that allow any third-party system to communicate with the PCC2.

Chapter 10

Final Remarks

This chapter identifies and briefly discusses some research related to what we have done, presents the contributions of our work and draws our main conclusions. Furthermore, it discusses which topics require further investigations.

This chapter is structured as follows:

Section 10.1 elaborates on related work.

Section 10.2 presents the conclusions and the contributions of our work.

Section 10.3 discusses how we improved the PCC.

Section 10.4 discusses future work.

10.1 Related Work

At the time of writing the research proposed by JiZhe [28] is the only research we found that also propose a ESB architecture based on Microsoft .NET framework. Their main contributions are a high level description of ESBs and a brief discussion on how they used WCF for parts of their ESB, but their research does not go into any details about their architecture and implementation.

However, based on their high level description we can conclude that their ESB is a lot less extensive and flexible than ours, because it does not provide a flexible management application and it does not provide functionality to host additional endpoints without having to redeploy their ESB. They also use WCF to communicate with Web services, but they do not describe how their ESB allows these Web services to communicate with each other, which is one of the most important principles of an ESB system.

Their research provides ideas on how to implement a custom ESB architecture using the .NET framework, but our research actually discusses an architecture and implementation for a flexible custom ESB architecture in detail.

The research proposed by Ji-chen [29] focuses on the ESB's concept, data's life-cycle and provides an overview of what tasks an ESB should perform, but it does not provide an architecture, like ours that describes how an ESB can be designed to actually perform these tasks.

The research proposed by Roshen [30] describes that when using currently available ESBs, each application must connect to the ESB through a port of a specific type, which is determined by the communication protocol and message type used by an application.

Their research proposes a new USB-like Universal Port, which can be employed by an ESB to allow any application to connect to an ESB, irrespective of the communication protocol or message format type used by an application.

While we think this is an very interesting idea, we did not use this approach because their research is still only theoretical, and as far as we know has not yet been used in any production ESB system.

10.2 General Conclusions

Commercially available ESBs can be a suitable choice for large companies, but it is possible that such a solution is inflexible, too expensive or too complex, and can introduce too much overhead for smaller companies.

To provide a solution for this problem we have identified the following two objectives for our research in Section 1.2:
(1) Identify the architectural principles for developing ESB systems with stringent flexibility and quality requirements.
(2) Define a custom ESB architecture, and evaluate this architecture by means of a prototype.

Using CompanyX and the PCC allowed us to analyze an ESB in a production environment to identify its limitations, which we used to capture the requirements of the PCC2.

We analyzed the architectures and documentation of the existing systems to identify the architectural principles for developing ESB systems. We then used these principles together with the PCC limitations to define the flexible custom ESB architecture of the PCC2.

We concluded, based on our analysis of the existing ESB systems that most ESB systems are Java-based, however our implementation shows that the .NET framework also provides technologies that facilitate the implantation of ESB systems.

We identified Windows Workflow Foundation 4.5 (WF4.5) as an interesting new technology for the development of ESB systems because it significantly increased the flexibility of the PCC2, and also simplified its development. This is because WF4.5 provides an extensive visual designer for the graphical construction of workflows that perform tasks such as message processing and service composition.

At the time of writing the PCC2 is the only ESB that we could find that already makes use of WF4.5, because it is a new technology that was released during the timeframe of our project. We are convinced that WF4.5 will be used by future ESB systems, because it looks very promising.

To evaluate the PCC2, we used the requirements of the PCC2 as evaluation criteria, and performed a case study.

The requirements evaluation results showed that the PCC2 fulfils all mandatory requirements, and satisfies the needs of the identified stakeholders by addressing the PCC limitations.

The case study evaluation results showed that the PCC2 is faster, and more flexible than the PCC, because it processes messages more efficiently, and facilitates the development of reusable adapters an workflows that can be deployed, configured and updated at runtime without downtime.

We proposed in this research a flexible custom ESB system, and demonstrated, using CompanyX as an example, that our flexible custom ESB reduces the time and cost of integrating third-party services.

Our work also shows that developing a custom ESB solution can be a suitable choice for companies for which the commercially available ESBs are not a suitable solution. The reason for this is that the PCC2 provides CompanyX with a solution that gives them the functionality they need, while it requires less resources. In addition, it is less complex and the development cost is less than the purchase price of the commercially usable ESBs that we researched.

We are of the meaning that the flexible architecture of the PCC2 can provide a solution for many more companies for which the commercial available ESB are not a suitable choice.

10.3 PCC Update

Besides our work on the PCC2, we have also worked on improving the PCC because in the time frame of our project the PCC message load increased significantly, which caused problems that prevented the PCC from operating correctly.

We identified the cause of these problems and determined that in order to solve them the PCC and its clients required an update.

Updating the PCC clients costs a lot of time and money because they run on the customers' computers, and installing and updating them has to be done by CompanyX's consultants. This is due to often occurring problems with the configuration of the customers' firewalls.

As part of the update we modified the PCC to provide a simplified version of the message box mechanism that we have designed for the PCC2. This mechanism prevents message loss and provides support for asynchronous messaging without customers having to open inbound ports in their firewalls.

Adding this functionality allowed us to significantly simplify the transition from PCC to PCC2 by preventing another update of the clients, and allowed us to already evaluate part of our architecture in realistic situations.

The updated PCC and its clients have been tested and approved by CompanyX's test department, and are currently being used by more than a hundred customers.

CompanyX's service desk employees confirmed that the update significantly improved the PCC performance, and solved the critical problems.

CompanyX's consultants reported that because the updated clients do not require any open inbound ports in the customer's firewalls the installation went smooty, which saved a lot of time and money because the updated clients can now be installed by customers instead of consultants.

10.4 Future Work

The architecture we provided leaves room for further investigation. In this section we describe the most important issues that can be tackled in future work:

USB-like Universal Port Analysis. It would be interesting to extend our adapter framework to provide an Universal Port type that delegates the actual communication to the correct adapter, and research if this approach is more beneficial than the approach we are currently using.

BPEL Import. Some companies use BPEL, which is a Business Process Execution Language, for the specification of business processes with Web services. To allow these companies to easily migrate their business processes to the PCC2, further research is required.

A starting point could be a tool called BPEL for WF [31], which is provided by Microsoft and provides import and export features between BPEL and WF. Further research should also determine how this functionality can be integrated into the PCC2 architecture.

Message Mediation. Our adapter framework provides security and communication protocol mediation to allow the workflows to interact with all endpoints by only using our internal message format. However, the developers still have to write the required code to describe the transformation that needs to be performed on the content of these messages.

Further research can improve the PCC2 by defining a mechanism to perform this transformation using a graphical user interface without having to define this transformation using code. Our workflow framework perfectly lends itself for such a mechanism, because WF4 makes it possible to develop custom activities that can provide the interface to do this. An example of a simple version of such a mechanism can be found in [32].

To perfectly integrate this mechanism, the PCC2 management application can be extended to allow managers to create, modify, and monitor workflows. This is possible because the Windows Workflow Designer can be rehosted in environments outside of Visual Studio 2012. This makes it possible to use the management application to develop a new transformation workflow using the custom actives to perform a transformation without any external tools, and without having to write any code.

Chapter A

Appendix

A.1 Optional Requirements

This section describes the requirements that we have specified based on the interviews with the identified stakeholders.

OR9: The system must only accept predefined message types from authorized external entities.
⋄ *Stakeholders:* Developers.
▹ *W7: No authorization for different message types.*
It must be possible to specify a list of registered message types that an registered external entity is allowed to receive and/or send.

OR10: The system has to provide support for a fallback address for external entities.
⋄ *Stakeholders:* Developers.
The fallback address has to be used when the system is unable to send the message to the main address after a specified number of times.

OR11: The system has to be able to act as an intermediary for message traffic between third-party services and internal services to provide a centralized location for logging, security, management and monitoring.
⋄ *Stakeholders:* Developers.
The system has to be able to dynamically host an endpoint with the same contract as a specific internal service. It should then log the incoming messages, check if the sender is authorized and authenticated to use this service based on his certificate and finally forward the message to the actual internal service.

OR12: The system has to be able to prioritize a single message.
⋄ *Stakeholders:* Service desk employees.
The system must make it possible to assign a high priority to a single message so

that it will be processed immediately after the message which is being processed.

OR13: The system has to log the processing times of messages.
⋄ *Stakeholders:* Developers.
The system has to log the time it takes to process a message with a specific message type so the average processing times can be calculated and be used to inform external entities about the processing time.

R14: The system has to provide the ability for message monitoring.
⋄ *Stakeholders:* Managers.
It must be possible to monitor incoming and outgoing messages by using the management application.

OR15: The system has to collect usage information.
⋄ *Stakeholders:* Managers.
It must be possible to view usage information that allows external entities to be billed for their usage of the system.

OR16: The system must be able to temporarily stop processing messages.
⋄ *Stakeholders:* Service desk employees.
It must be possible to temporarily stop processing messages. The system must still accept incoming messages but they have to be buffered until a manager indicates that the system must process the messages again.

OR17: The system must be able to temporarily block external entities.
⋄ *Stakeholders:* Managers.
It must be possible to temporarily block external entities so that the system will not accept any messages from that external entity.

OR18: The system must be able to process messages based on the priority of the message type.
⋄ *Stakeholders:* Managers, External Entities.
It must be possible to prioritize the supported message types which allows the system to process incoming messages based on this priority.

OR19: The system should be able to process messages based on the priority of the external entity.
⋄ *Stakeholders:* Managers, External Entities..
It must be possible to prioritize external entities which allow the system to process incoming messages based on the priority of the external entities.

OR20: The system must send problem alerts.
⋄ *Stakeholders:* Managers.
The system should send email alerts when a third party service has technical difficulties.

OR21: The system has to be able to generate reports based on the specific content of messages.
⋄ *Stakeholders:* Managers.
It must be possible to generate and view reports based on specific content of a certain message type.

A.2 Message Flow

Below a brief description is given on how the PCC processes messages that are received from a client application.

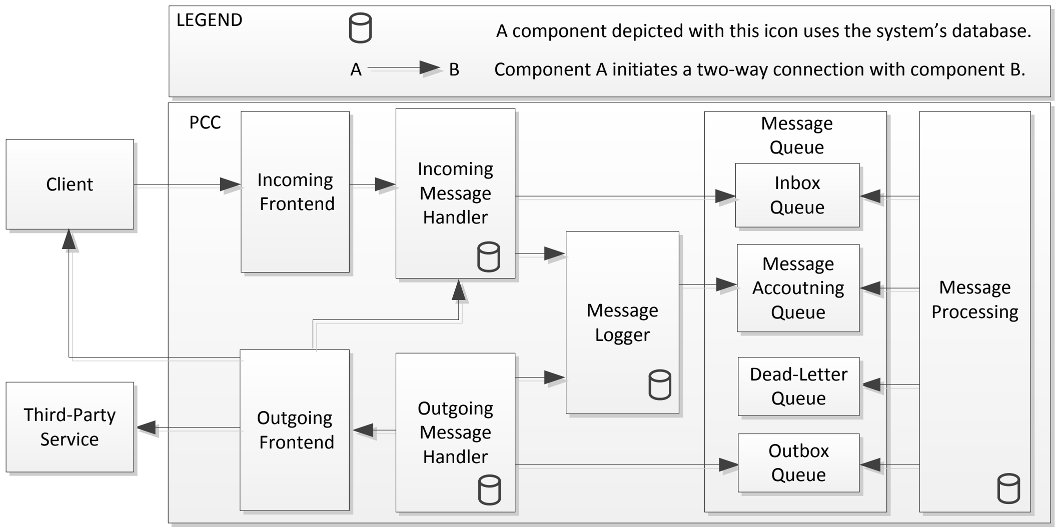


Figure A.1: PCC Present Architecture

- A client application sends a message to the Incoming SOAP Frontend.
- The Incoming SOAP Frontend keeps the connection with the client application open and forwards the message to the Incoming Message Handler.
- The Incoming Message Handler stores the message in the inbox of the Message Queue and sends the message to the Message Accounting service which logs the message to the database.
- The Incoming Message Handler then notifies the Incoming SOAP Frontend that the message was successfully stored.
- The Incoming SOAP Frontend, which still has an open connection with the client application, then sends a synchronous reply back to the client application to indicate that the message was accepted by the system.
- The Message Processing service detects that there is a new message in the inbox of the Message Queue, it then processes the message, adds the destination address to the message and stores the result in the outbox of the Message Queue.
- The Outgoing Message Handler detects that there is a new message in the

outbox of the Message Queue and forwards it to the Outgoing SOAP Frontend.

- Based on the added destination address the Outgoing SOAP Frontend forwards the message to the correct third-party service, and synchronously receives a reply message which it forwards to the Incoming Message Handler.
- The Incoming Message Handler then stores the message in the inbox of the Message Queue and notifies the Outgoing SOAP Frontend that the message was successfully stored.
- The Outgoing SOAP Frontend then notifies the Outgoing Message Handler that the message was successfully processed so it will delete the message from the Message Queue.
- Finally the Message Processing service detects the reply message from the third-party service, processes it, and stores the result in the Message Queue's outbox. The resulting message is then sent to the client application via the Outgoing Message Handler and the Outgoing SOAP Frontend like the original message.

Bibliography

[1] David A. Chappell, *OReilly - Enterprise Service Bus*. O'Reilly, 2004.

[2] C. de Hullu, "Evaluating .net-based enterprise service bus solutions," 2012.

[3] M. A. Akif, "Microsoft's Enterprise Service Bus (ESB) Strategy," 2006.

[4] D. Hatzidakis, "To ESB or Not to ESB? ESB - An "Architectural Pattern" ," *October*, 2006.

[5] R. Gupta, "Enterprise service bus capabilities comparison," *Business*, no. April 2008, 2008.

[6] J. Enterprise and S. Bus, "Open Source Enterprise Service Bus (ESB) Evaluation Vendors on their ESBs," 2009.

[7] A. Rahien, "Rhino service bus," December 2008. Online: `http://ayende.com/blog/3752/rhino-service-bus`.

[8] "Phoenix service bus." Online: `http://pservicebus.codeplex.com/`.

[9] "Mass transit." Online: `http://code.google.com/p/masstransit/`.

[10] "Simple service bus." Online: `http://simpleservicebus.codeplex.com/`.

[11] "Shuttle esb." Online: `http://shuttle.codeplex.com/releases/view/61788`.

[12] "Nservicebus." Online: `http://nservicebus.com/`.

[13] "Esb.net." Online: `http://keystrokeesbnet.codeplex.com/`.

[14] "Nginn.messagebus." Online: `http://code.google.com/p/nginn-messagebus/`.

[15] "Neuron esb documentation." Online: `http://products.neudesic.com/downloadpage`.

[16] Neudesic, "Neuron esb: An enterprise service bus for the microsoft platform part 1," 2010.

[17] Neudesic, "Neuron esb: An enterprise service bus for the microsoft platform part 2," 2010.

[18] D. Chappell, "Understanding BizTalk Server 2006," no. August 2005, 2006.

[19] "Biztalk server documentation." Online: `http://www.microsoft.com/biztalk/en/us/product-documentation.aspx`.

[20] M. Corporation, "Microsoft biztalk server 2010 technical overview,"

[21] "Biztalk server adapter page." Online: `http://msdn.microsoft.com/en-us/library/aa546748(v=BTS.70).aspx`.

[22] D. Chappell, "Introducing Windows Communication Foundation," *Framework*, no. January, 2010.

[23] C. . A. David Chappell, "Introducing windows communication foundation in .net framework 4," tech. rep., Copyright Microsoft Corporation, March 2010. Online: `http://msdn.microsoft.com/library/ee958158.aspx`.

[24] D. Chappell, "The workflow way: Understanding windows workflow foundation," 2009.

[25] “Windows forms.” Online: http://msdn.microsoft.com/en-us/library/dd30h2yb.aspx.

[26] M. Fowler, *Patterns of Enterprise Application Architecture.* Addison-Wesley Professional, 2002.

[27] “Managed extensibility framework.” Online: http://msdn.microsoft.com/en-us/library/dd460648.aspx.

[28] Y. Y. Li JiZhe, “Research & implementation of lightweight esb with microsoft .net,” *International Conference on Frontier of Computer Science and Technology*, 2009.

[29] G. M. JIANG Ji-chen, “Enterprise service bus and an open source implementation,”

[30] W. Roshen, “Enterprise service bus with usb-like universal ports,” *2011 Ninth IEEE European Conference on Web Services*, 2011.

[31] “Bpel for windows workflow foundation.”

[32] R. Kiss, “Wf4 custom activities for message mediation,” Dec. 2012.