Microsoft

FREQUENTIS

Always-Responsive Multitier Solutions with .NET 3.5 SP1 (WCF, WPF)

Frequentis AG

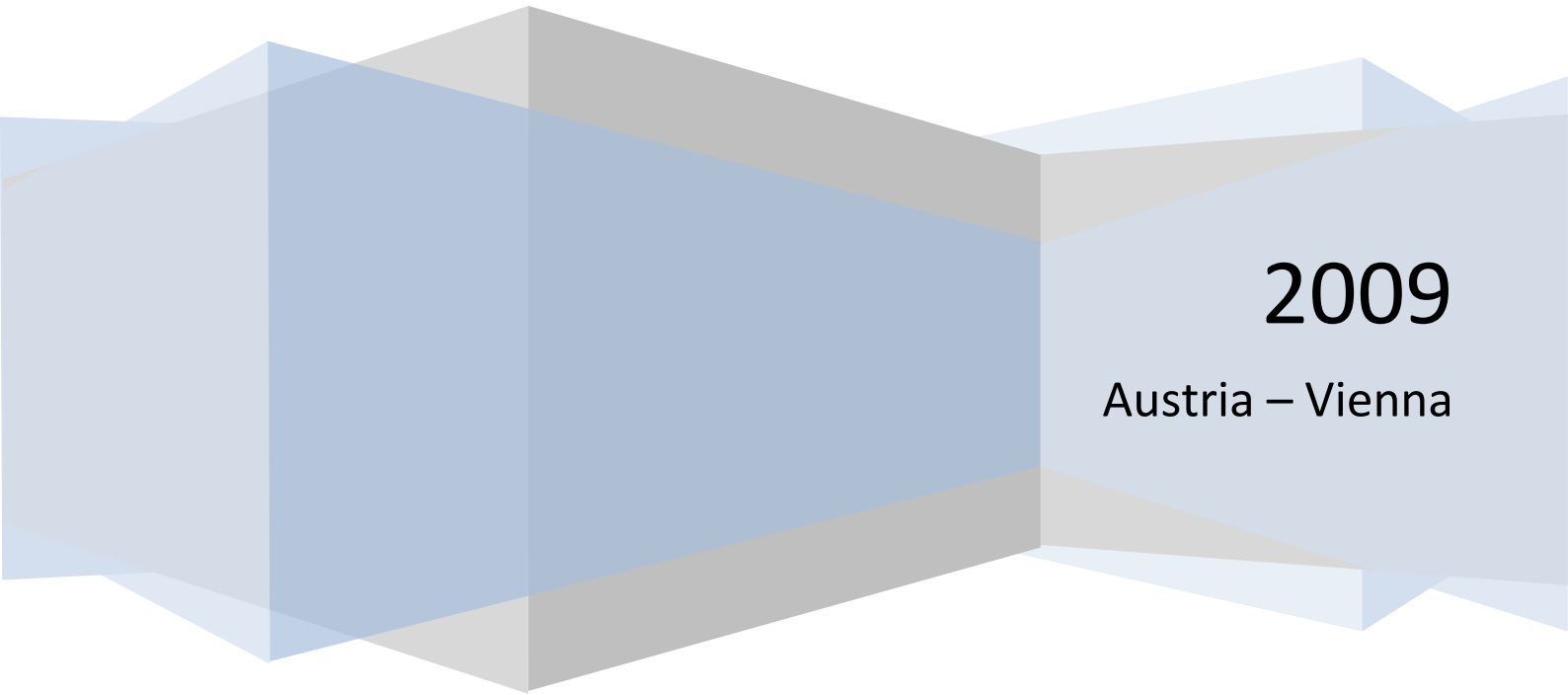
A Services-Layer and Smart Client for monitoring ship-navigation

Ulrich Hüttinger, Stefan Domnanovits (Frequentis AG)

Architects, Inventors of the Concepts described in this Whitepaper

Mario Szpuszta (Microsoft Austria)

Architect, Technical Writer of this Paper, Advisor for Frequentis in this Project



2009

Austria – Vienna

Table of Contents

Background-Information on the Customer and the Project

Architecting, designing and implementing solutions and applications for public safety is a very special and hard challenge – independent of the technology choice you're taking. These kinds of applications need to be "always responsive", they need to fulfill hard performance requirements in many cases and failure of parts of the system may not lead to complete system outages at all!

Dealing with these challenges is all about driving appropriate architectural decisions at the very beginning of the project. These decisions are much more important than any subsequent technology or implementation decision and this is exactly the experience we've had to make together with Frequentis AG in a very interesting project on securing ship vessel traffic for large ports/havens.

Frequentis AG has its origins in securing air traffic where they built special, reliable and robust communication devices and systems for ground-to-ground, ground-to-plane and plane-to-plane communication. More and more these devices are extended with systems for dealing with, monitoring and managing emergency cases in a variety of other areas in addition to air traffic such as fire fighting, police and ship vessel traffic.

This whitepaper is all about discussing the architectural decisions and motivations for these decisions made by Frequentis AG in their Tracking and Tracing (TnT) project for monitoring and managing ship vessel traffic in large havens/ports. The idea of this paper is giving you some aspects and thoughts in your hand on why, when and how you can leverage the .NET Framework 3.5, especially WPF and WCF, for structuring highly available services and client applications. Within this whitepaper we are giving you some background information on the requirements Frequentis had to deal with. Then we will discuss about some of the most important architectural decisions and their motivation to give you an architectural understanding of how-to leverage WCF/WPF for the kinds of situations described in this paper.

Note: this paper is about architectural decisions, not about implementation details. While we discuss these architectural decisions in the context of Microsoft technologies such as WPF, WCF and the .NET Framework 3.5, the focus is not about how patterns have been implemented with these technologies. We rather talk about how these technologies fit into the larger architectural picture of the solution built from Frequentis and why Frequentis decided to leverage these technologies the way they are leveraging them!

Monitoring Ship Navigation – Tracking & Tracing

In this whitepaper we're discussing architectural decisions Frequentis had to make for a system called Tracking & Tracing which is a mission critical extension to the Frequentis Maritime Communication Systems (MCS). The Frequentis Maritime Communication System is a special, reliable and robust system for land-to-land, land-to-ship and ship-to-ship communication. The MCS is baked into several hardware-devices with controller-interfaces (typically several monitors with communication controller interfaces in front of it). The MCS has several extensibility and integration interfaces and the primary target of Frequentis is extending MCS with systems for monitoring and historically logging ship vessel traffic as well as collaboratively working on exceptional and even emergency cases.

Typically these kinds of applications consist of two parts – client-side applications which are running on so called **working positions** of employees which are responsible for monitoring ship vessel traffic and dealing with exceptions. These working positions are connected with the Maritime Communication System (MCS) itself which expresses itself as a rich communication device with several monitors and communication options for the employee on his working position, as well. On the other hand each working position needs to be connected with a variety of backend systems as well as with other working positions of other employees in the same or other locations so that these can work collaboratively on the same emergency case in an efficient way.

The following figure illustrates the typical situation that needs to be supported by the Tracking & Tracing system – please not that this figure does not reflect any architectural decision. It reflects a view on the situation that is **required** to be supported. Also note that it is a simplified view – think about much more working positions at several locations in real deployments.

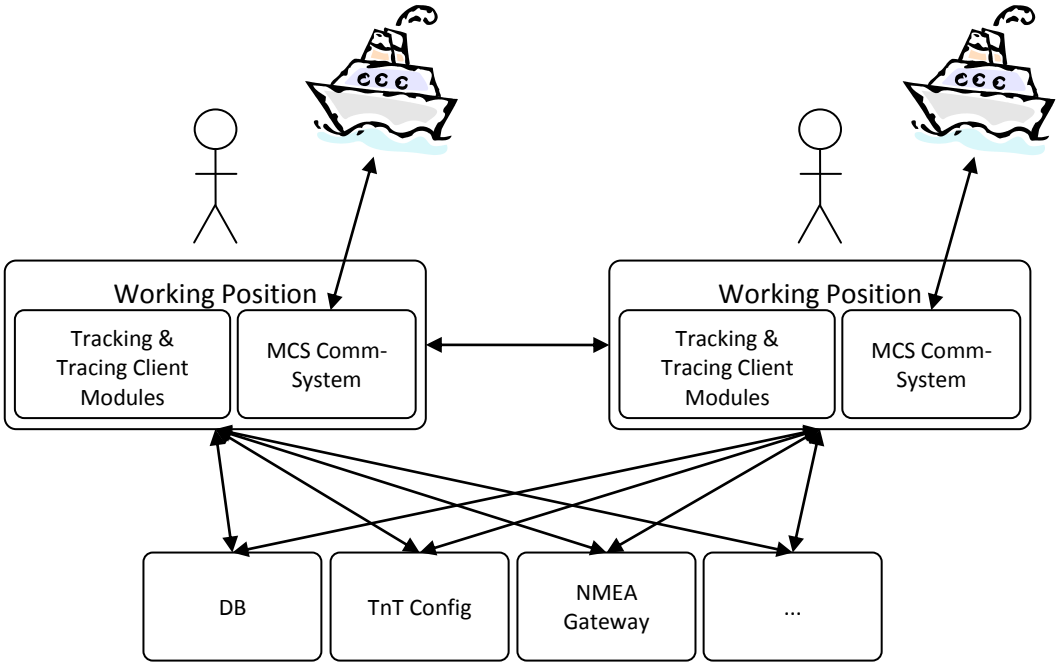


Figure 1: Communication requirements between working postions and backend systems

As you can see, for collaboratively monitoring ship-vessel traffic and working on emergency cases, the interconnections between the working positions of the employees of the same or even different

ports (havens) is extremely versatile but is required for quick and direct communication between the working parties!

Backend systems working positions are typically connected to include databases for the historical logging, configuration systems for release, patch and configuration management and business systems such as the AIS network which is using the NMEA-protocol standard for communication (for more details take a look at http://de.wikipedia.org/wiki/Automatic_Identification_System). NMEA is the abbreviation for National Marine Electronics Association (NMEA). The AIS-network enables communication between marine electronic devices such as depth finders, nautical chart plotters, navigational instruments or even engines and tank sensors and it can be used to get information on ships involved in emergency cases or any other exceptional cases. Of course there are lots of other systems the tracking and tracing solution needs to provide or be integrated with.

From the end-user's perspective, a typical working day of employees sitting at the aforementioned working positions involves working with different monitoring applications. These applications need to operate in a multi-monitor environment and they need to be as loosely coupled with other applications as possible for availability reasons (if one fails the other may not be interfered at all). Typically within these applications employees need to watch and monitor lots of incoming trace messages (typically retrieved through systems such as the NMEA gateway), analyzing them and forwarding instructions to ship vessels on changing their navigation or other behaviors. As the user needs to deal with a huge amount of messages per day where he needs to quickly differentiate between less important and really important messages for the historical logging as well as for effectively dealing with exceptional cases, the user interfaces itself of the running client tracking & tracing application needs to be intuitive, extremely easy to use and of course always responsive independent of the state of other working positions or even backend systems!

Technologies in scope of this Paper

Frequentis AG is using the following technologies for implementing the solution based on the Microsoft platform stack:

- For the communication between working positions and backend-systems, Frequentis uses **Windows Communication Foundation (WCF)** from the **.NET Framework 3.5 (incl. SP1)**
- New clients and modules that are plugged into the tracking & tracing infrastructure will be developed with **Windows Presentation Foundation (WPF)** of the **.NET Framework 3.5 SP1** due to the large amounts of data users have to deal with and the requirement of having smart, intelligent and extremely easy to use user interfaces.
- **Windows Server 2008** as application server for tracking & tracing backend services
- **SQL Server 2005** for capturing and archiving logging and historical information on ship traffic

Basic Architectural Decisions and Their Motivation

You should have enough information to understand the most important architectural decisions that Frequentis has been decided to follow during several internal discussions and conversations with Microsoft in Austria. Below I'll summarize the most important non-functional requirements that influenced the solutions architecture for the system:

- Intuitive user experience for effectively dealing with a large amount of critical and non-critical data on ship vessel traffic.
- Always responsive UI for collaboration within and without exceptional cases.
- High availability of the communication-platform for talking to backend-systems.
- Communication between working positions and backend systems into all directions.
- Fail-safe behavior of the communication stack. The communication between different working positions working on the same case as well as working positions and available backend systems needs to be possible at all the time without any interruptions – independent of the availability of any specific backend system or working position. That means if one or a few backend systems or working positions are down, the communication to and from other working positions and backend systems may not be interfered!

Taking a look at these requirements, an always available and failsafe communication stack as well as an always responsive UI are the core non-functional requirements which are present through all parts of the system. The core architecture of the system needs to support these non-functional requirements at all stages and layers. Within this whitepaper we will discuss the architectural decisions Frequentis had to make for leveraging WCF and WPF on the front-end and the backend-side for supporting these non-functional requirements!

Overall System Architecture & Motivation

Transforming the illustration in Figure 1 into a more technical diagram will result into the following conceptual view of the overall system landscape (note – *TnT* is the acronym for *Tracking & Tracing*):

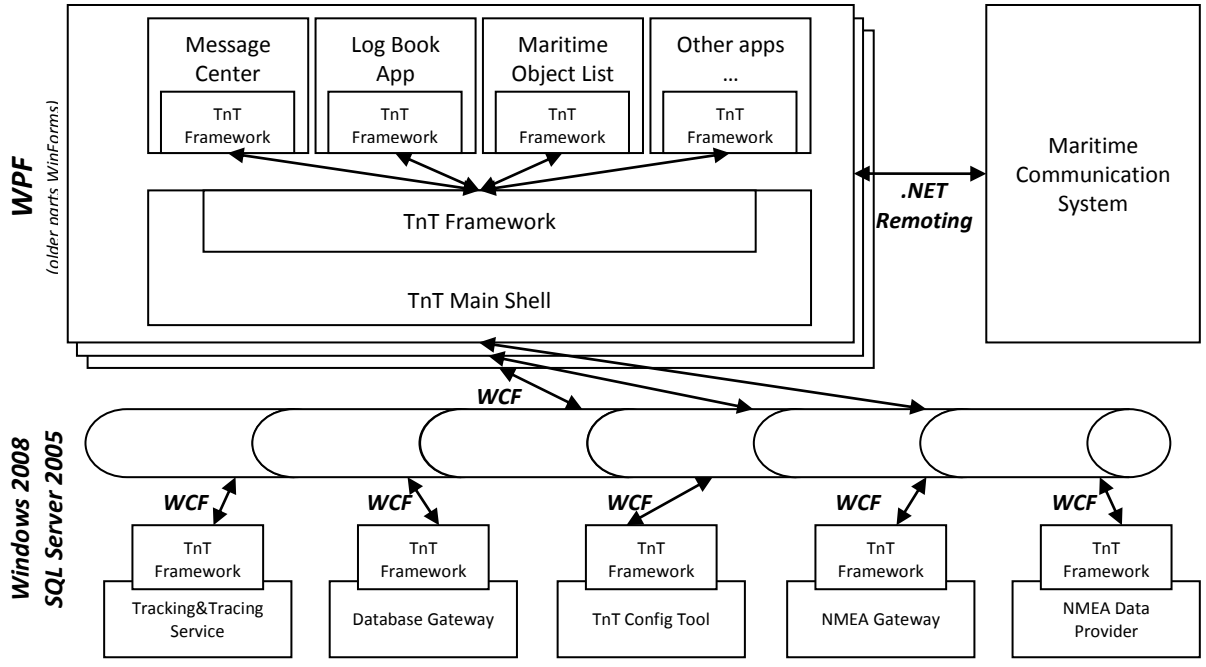


Figure 2: The Tracking and Tracing (TnT) solution architecture overview

As you can see, each working position is equipped with several independent client applications as well as a common “shell” running on the client that integrates all the different client applications in a loosely coupled fashion and that is dealing with communication to the MCS as well as backend systems. Each working position client application will be able to talk to the MCS, to other working positions or to central backend systems through a message-bus concept. I intentionally wrote message-bus “concept” as the real implementation is not really a message bus as of now as you will read in the subsequent two chapters of this chapter! The Log Book application is the first client implemented with Windows Presentation Foundation and some new UI paradigms in the whole TnT-landscape. The second part of this paper is dedicated to the WPF-based discussions as you will see later while reading through the paper.

Note that the communication to the Maritime Communication System (MCS) takes place through .NET Remoting. The MCS-system belongs to one of the core-products of Frequentis AG which is around for several years now. This system on its own consists of a fully-fledged front-end and backend system originally based on .NET Framework 1.x. The system offers interfaces based on .NET remoting on the client as well as the server. Tracking and Tracing applications of course need to be integrated with MCS as the TnT-systems need to get updates from radio channels managed by MCS itself for catching radio transmissions between ships and land-stations. Also this .NET Remoting interface is used for 3rd party integration into the MCS application – and TnT is using the same .NET Remoting interface as it is seen as a separate product with integration points into MCS.

Each backend service is made available through service interfaces implemented with Windows Communication Foundation (WCF). These service interfaces are using data transfer objects which are common for all TnT-applications. Each application and service is using these common set of objects to talk to other clients and services through the message-bus infrastructure.

All services on the backend are implemented in a stateless fashion so that these services can be scaled out and operated on a variety of machines in a load-balanced environment. Common for all services is an asynchronous processing pattern. That means, the defined data transfer objects (DTO) for the tracking and tracing environment are essentially commands with data payload, which are sent from working positions to services or other working positions and vice versa. The services (or other working positions) are then processing the commands and the data payload transferred within the DTO. As soon as a service (or other working position) has finished its task it pushes the result message back to the message-bus infrastructure and working positions which are interested in the result can process it afterwards. That means, the whole system does not have any classic request-/response based service interface at all and every party retrieves messages through its own service interface in an asynchronous fashion. This concept is core for ensuring high responsiveness throughout all parts of the system.

A single client essentially is a working position that runs multiple, slim client applications on it which are used by operators on several monitors. While these applications are separate executables running on the client running in an independent and isolated fashion, they are all launched from a central client-application called TnT Main shell. Having separate EXE-clients first of all makes the different applications more independent of each other and avoids side-effects of one application to other applications running on the same working position (compared to plug-in based scenarios where such a strong isolation within a single process is not possible with current versions of the .NET Framework). Furthermore it makes different licensing and deployment-scenarios easier to

implement. Communication to the message-bus infrastructure also happens through TnT Main shell which is always running in the background and replacing the classic Windows shell on the working positions. The TnT Main application leverages WCF client channels and the data transfer objects defined by Frequentis for the TnT-environment to talk to other participants such as backend services or other working positions through the message bus infrastructure. It is important to note, that the communication patters for exchanging messages between each TnT client application and the TnT main shell are exactly the same as the ones are used for communication between the working positions and backend services or even other working positions. The communication between TnT client applications and TnT main shell on a single working position just uses other protocols and bindings but is asynchronous as well which is core for ensuring availability and responsibility of the client applications themselves, as well.

These asynchronous communication patterns, the base classes for leveraging them as well as the interfaces for commands / jobs and their execution are part of the TnT framework which is reused across all actors in the overall system architecture. The data transfer objects used for sending and receiving commands and data are part of this framework. One of the most interesting aspects in this area is the fact that each application within the TnT landscape ships with its own data transfer objects. DTOs are not shared as “shared library assemblies” between applications. They are shared as contracts at the wire-level based on data contract and data member names, only! This is primarily driven for decoupling the applications and especially the distributed development teams around the world. Whenever one application wants to reuse DTOs of another application, the development team re-builds exactly those parts of the DTOs they require for their own application by building WCF data contracts using the same names for the contracts themselves and the same member-names (of only those members they really need) in their own DTO-libraries. That means the systems within the TnT landscape are really loosely coupled so that changes from one application do not affect other applications at all. Also it makes the distributed development scenario easier to handle from a project-management and guidelines perspective as different projects which are responsible for different application parts in the TnT landscape do only talk about wire-contracts and no single shared-code part! The same approach is also used for third-party application integration where the decoupling between the systems and keeping systems more independent of each other is even more important.

Technical detail besides: Frequentis had to use the DataContractSerializer of WCF instead of the NetDataContractSerializer to make this work because the second one serializes assembly version and assembly namespace information onto the wire which isn’t really good for interoperability and loose coupling scenarios.

Finally the standardized service interfaces which are used by working positions as well as backend services for receiving and sending DTOs for transmitting instructions and data to other actors are parts of this framework.

Logical Layers in typical TnT-Applications

The overall architectural representation of each application plugged into the Tracking & Tracing landscape always results into the same architectural approaches outlined in the following figure:

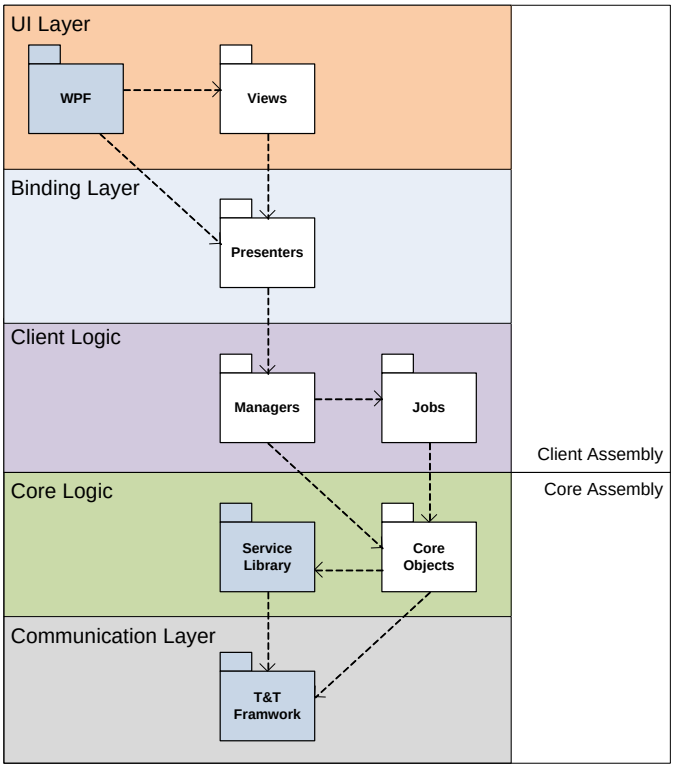


Figure 3: General approach for a layered client architecture

In the UI layer and the binding layer you will find typical implementations of UI-focused patterns such as model-view-controller, model-view-presenter or the new presentation model as discussed in chapter Decisions Specifically Tailored to WPF on page 18 in this paper. The presenters- and controller-classes then access a core client logic which is responsible for asynchronously executing client-side jobs (items) as discussed and shown in the chapter Connecting the Front-End with the Business-Logic Layer on page 23.

The core logic and the communication layer packages are then part of an overall framework used for all client- and service-applications in the TnT landscape as you can see in Figure 2 earlier. These libraries are used for asynchronous execution of jobs that encapsulated business logic. Functionality implemented in these libraries can be used on both, the client-layer and the backend-layer. These packages include objects such as interfaces, data-transfer-objects (DTO) for talking to backend-services with WCF, implementations of queues for asynchronously processing jobs etc. Architectural approaches embedded into these framework-level components are discussed in the chapters Decisions for the Message Bus Infrastructure with WCF (page 10) as well as Decisions on Asynchronous Command and Message Processing (page 13) in this whitepaper.

Decisions for the Message Bus Infrastructure with WCF

For establishing a flexible, reliable and always available communication between several working positions as well as between working positions and services, an infrastructure for dealing with this communication is necessary. This is the task of the message-bus infrastructure in our solutions architecture.

Reviewing the communication requirements, Frequentis had to deal with the following aspects in their infrastructure:

- Complex network – reflecting Figure 1 essentially every actor should be able to send and receive messages to any other actor.
- Always available network – outages of one part of a system may not affect the communication infrastructure at all. If an actor is unable to receive a message, other actors should continue to work and continue talking to any actor who is still working.
- Multiple receivers for single senders – it might be possible that multiple actors in the communication network are interested in messages sent by one or more senders of messages depending on the message type and its contents.

While the messaging-infrastructure is implemented using **Windows Communication Foundation** (WCF) as a technology, there are multiple communication-patterns and styles for establishing the communication between all the different actors. In WCF these patterns and styles are supported by selected appropriate **bindings** and **behaviors**. Frequentis had to (and still has to) deal with several approaches for providing this messaging infrastructure as you will read in the following two chapters.

Peer-to-Peer Communication as a Message Bus with WCF

Let's review the communication-situation for the TnT system – many actors are talking to many other actors in an asynchronous fashion. Actors are sending messages multiple other actors might be interested in and vice versa! Furthermore the message-bus infrastructure needs to be as available as possible and the failure of one or more nodes involved in the communication should not lead to a failure of the overall communication.

This is a perfect scenario for peer-2-peer based communication clouds because peer-2-peer communication does not rely on a central communication broker which would break the whole communication if it finally fails. In a peer-2-peer based system, the communication network is established through all nodes involved in the cloud and if one or more nodes fail, the other nodes in the same cloud could still talk to the other available nodes.

Finally, these characteristics of peer-2-peer communication perfectly address the aforementioned communication-needs for the TnT system of Frequentis and the TnT-team selected the available bindings and behaviors of WCF to establish the message-bus system. By using these bindings, it was possible establishing a stable communication cloud without having a central point-of-failure (a communication broker) and the need of running a separate infrastructure for communication between the different parties.

Simply by evaluating contents of data transfer objects and message headers a node such as a backend service or a working position could evaluate, whether a message was interesting for it or not.

But when doing several performance tests with the peer-2-peer based bindings provided by WCF since its first version, Frequentis unfortunately uncovered performance issues with the peer-2-peer communication channels. Peer-2-peer communication was originally not built for systems where thousands of messages are sent across the wire within a very small period of time (below a second). With these bindings actors could send or receive a maximum of 700-1000 messages per second whereas the requirement for the TnT-landscape is the capability of processing over thousands of messages per second.

This is sufficient for most of the typical environments where TnT will be used, but there are some large-scale environments where a different approach needs to be applied as the performance requirements will definitely be higher! Therefore Frequentis needed to think about an additional alternative to the peer-2-peer based communication with WCF!

Pragmatic Brokered Message Bus Architecture with WCF as Alternative

If a peer-2-peer cloud cannot be established to ensure communication between the different working positions and backend services, establishing a central communication broker infrastructure is the only approach you can implement with currently available out-of-the-box technologies and products of the .NET Framework. And this is exactly what Frequentis will do for the huger deployments of their TnT-system.

In this approach, the message bus needs to be implemented as a separate service within the overall system architecture. Essentially this service will have to manage publishing of messages to all interested actors as well as the interested actors themselves by managing subscriptions. So with this approach we will have to implement a classic publish-/subscribe pattern in a very pragmatic and simple fashion. The ideas we've been coming up with and that we will follow in the future are the following ones:

- Every actor will need to provide an endpoint for receiving messages. This also applies to the peer-2-peer binding using the appropriate peer-2-peer configuration in WCF. The only difference is that working positions and backend services need to provide these endpoints with a different binding such as net.tcp, basicHttp or wsHttp.
- Each actor that is interested in receiving messages with certain contents or commands will need to tell this to the communication broker by subscribing to these messages. The communication broker will store this subscription information into a database. The subscription information essentially consists of an endpoint URI (a WCF endpoint) as well as for which messages a subscriber is interested in.
- All actors need to send messages through one of the message broker endpoints. The message broker then evaluates the contents of the message, matches these contents with information stored in its subscription database and the forwards the message to each endpoint with fitting criteria by performing a WCF call to the endpoint stored in the subscription database for the interested actor.

For implementing such a broker, we technically do not need more than a Windows Server that runs the WCF broker services as well as a database that stores the subscription information.

But what's about reliability and availability then? If everything of the broker runs on a single machine, the communication network will break if this machine is down.

And finally we have several thoughts on mitigating this risk:

- First we can run the broker services in a farm of machines all using the same SQL Server Database running in a failover cluster for managing subscriptions.
- Of course it is possible running several of these farms. E.g. a possible deployment would be having four farms whereas each farm is equipped with one load balancer to split load between the machines in one farm as well as three or four cheap web service machines that are hosting the WCF broker service for the TnT system. Clients could then be configured with alternate endpoints of broker (farms) and if they're unable to reach one broker farm they could try to contact the next farm.
- As a last fallback we could still establish a peer-2-peer cloud with WCF if all brokers or broker-farms as outlined right before are down. In such a case the performance would be slower but dealing with a performance penalty is still better than dealing with an outage of a system in an emergency case.

Essentially the implementation of the actual WCF services as well as the service contracts are not very much affected by these architectural changes as the bindings and protocols in WCF are independent from the implementation of these services and interfaces and are essentially configured through the application configuration. The following figure illustrates currently arising ideas for a message-bus infrastructure in the TnT-landscape (simplified, sample communication):

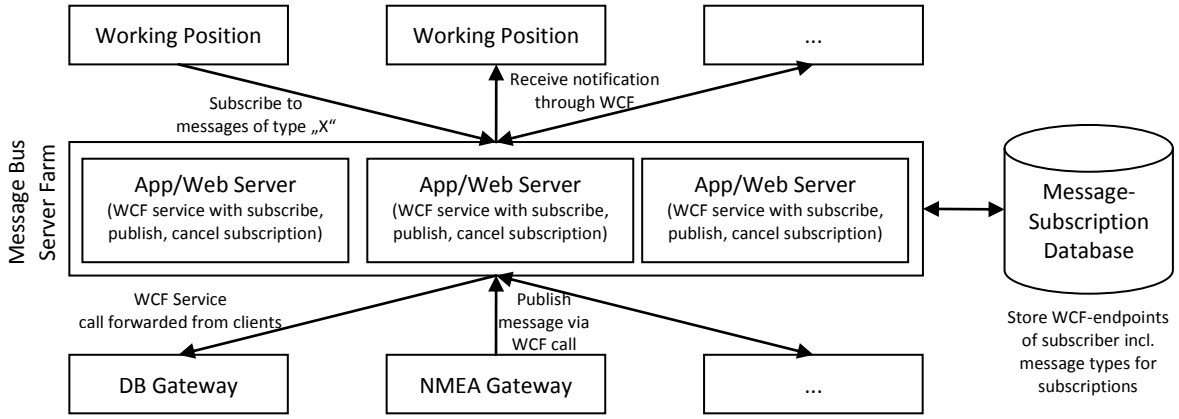


Figure 4: A pragmatic message-bus architecture for the TnT-landscape

The only part that needs to be modified in its implementation to be able to support both scenarios – the peer-2-peer cloud and the brokered communication – are the proxies / agents that are used for talking to the backend services or other working positions. These agents need to be aware of whether they need to publish messages by just publishing them to the peer-2-peer cloud or by calling endpoints of the message broker. And if a message broker endpoint is not available, these agents need to implement the fallback scenarios by publishing messages to a peer-2-peer cloud or to alternate endpoints of message brokers. But all the other parts including service provider endpoints and interfaces can mostly remain as they are for both approaches due to the plug-able architecture of the WCF infrastructure, especially as Frequentis is using asynchronous patterns throughout the whole solution, already.

Decisions on Asynchronous Command and Message Processing

Processing messages and commands asynchronously is a core strategy for keeping the system responsive on both, the client and the server-side! For providing the same level of asynchronous message processing at all levels for executing business logic, Frequentis implemented the core interfaces and base classes for asynchronous message processing in the TnT Framework (Figure 2).

Note that based on this information the following architectural concepts can apply to each participant in the TnT system architecture. Business logic based on request messages can be executed on clients in WPF-applications or Windows Forms applications (for older parts) where views and other components are sources of requests as well as within services where WCF-facades and endpoints lead to the execution of business logic in the same, asynchronous manner.

Also many service-providers, including services running on the backend or other clients accepting messages from other actors in the communication cloud, need to be able to scale within one executing instance and not just across multiple machines. Therefore these service-providers must be able to process multiple requests at the same time if necessary. For this purpose each service with the requirement of executing multiple requests at the same time needs to manage a pool of execution threads which are processing incoming messages.

In this and the subsequent chapter we will discuss the most important design decisions Frequentis made for fulfilling these kinds of requirements.

Job-Pattern with Commands as Requests and Responses

In my opinion, one of the most important concepts for implementing a scalable and always responsive architecture is the concept of a job or a task. This concept is implemented in Microsoft-assets for concurrent processing such as the Concurrency and Coordination runtime originally implemented for the Microsoft Robotics Studio product, as well!

Frequentis decided to use the Job design-pattern for implementing the core business logic of their application (note, they do not use CCR mentioned above, they have their own implementation based on .NET – one of the main reasons for that way the fact that when the implementation started, CCR was still not available as a separate download and not officially supported for commercial projects – both is the case now).

That means all the business logic provided across the different parts of the application is implemented in **Job-classes** which are all implementing the **IJob**-interface. Jobs are primarily executed by modules which are receiving commands. If a job is executed in the client, commands are received through an interface in-process. But at the same time it is possible to receive commands as a service through a WCF-endpoints and message-interfaces. In any case the **Command**-objects implementing the **ICommand**-interface are received through a message gateway. Commands are ordinary classes with some properties and data contents sent within request-messages to these gateways. These commands are translated into one or several jobs by a **CommandProcessor**-object (which in turn implements the **ICommandProcessor** interface).

The following figure illustrates the main ideas of the asynchronous job execution engine at a conceptual level:

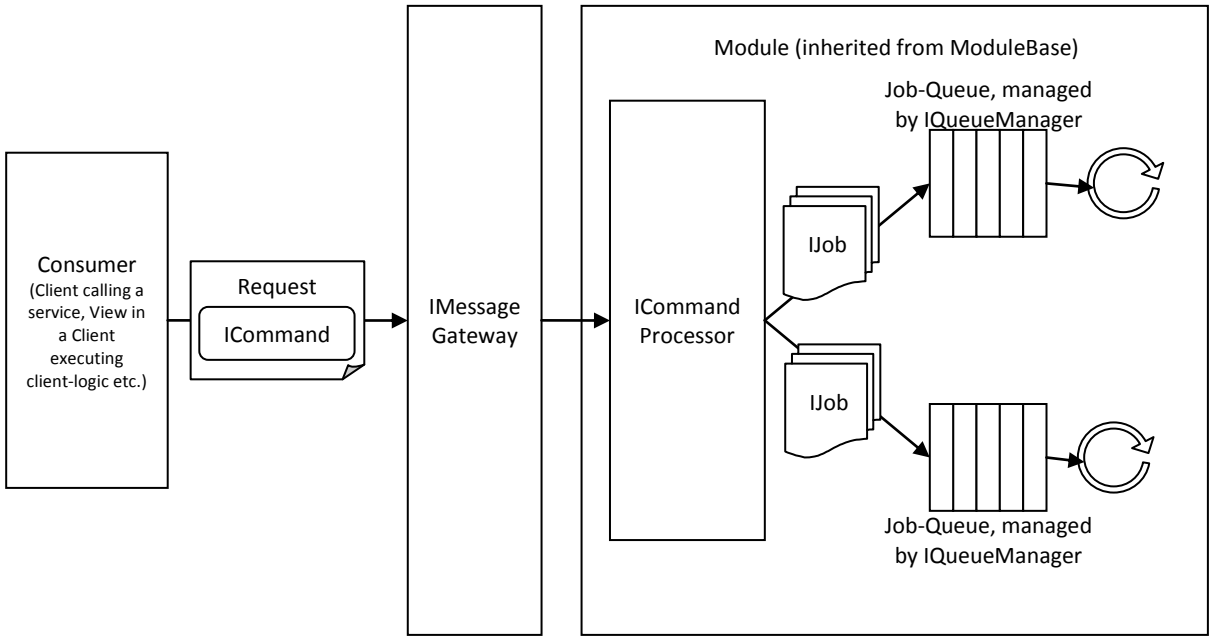


Figure 5: Commands, Jobs, Queues and Execution-Threads

As you can see a message gateway (either hosted in the client or in a service at the backend) receives requests containing commands. The gateway is nothing else than a class that inherits from **ModuleBase** which then forwards commands from receiving messages to command-processor classes. These command-processors are responsible for mapping commands to jobs and forwarding those to queue managers which are in turn responsible for executing these jobs. The jobs contain the actual business logic.

Jobs are always put into working queues. Each queue is queried by a pool of executing threads which in turn pick jobs from these queues in a first-in-first-out strategy and execute these jobs in parallel. The thread pools are implemented based on ordinary .NET Framework thread pools using the ThreadPool-class of the System.Threading namespace from the .NET Framework.

Queues, QueueManager and Worker Queues

Please note in Figure 5 that the queues for queuing jobs are managed by a separate component implementing the **IQueueManager**-interface. The QueueManager can work with and manage several queues for processing job and he is responsible for executing the jobs in the actual worker threads from these queues. For each type of job, the QueueManager creates separate queues. This enables the queue manager differentiating types of queues and processing strategies for different types of jobs.

In the current implementation the QueueManager is able to deal with two types of queues, **WorkerQueue** and **PooledWorkerQueue**, created depending on the job-type. These queues are executing jobs within their own thread or threads (depending on the type). Both queues are using simple System.Collection.Generic.Queue<type of jobs> objects for implementing the core queue-functionality and both queue-types are responsible for managing the threads for job-execution. So don't be afraid, there is no complex infrastructure such as MSMQ in the game with this implementation. You can implement all the patterns discussed based on ordinary .NET classes. The

difference between the two types of queues essentially is the way they are managing the threads for executing jobs in their queues.

A **WorkerQueue** implementation has a single thread that it uses for executing jobs in its Queue<T> and therefore is suited for jobs that need to be executed one after another exactly in the order they've been put into the queue. On the other hand the **PooledWorkerQueue** uses the .NET ThreadPool class for managing a pool of threads for executing jobs. That means it has several threads which are executing multiple jobs at the same time and can be used for jobs where the order of execution is not relevant.

How Jobs return Information to their Caller

Whether a module is hosted within a client application on a working position and executing business logic jobs or on a service at the backend-layer, it may communicate results from processing its job back to its callers.

Typically such notifications with results are sent back via event notifications containing response parameters from the module contacted through a message gateway by the original caller (remember – ModuleBase in Figure 5). For this purpose Frequentis invented the **INotify**-interface which is implemented by the **ModuleBase**-class which on its own represents or IS the actual message gateway. INotify defines methods intended as callbacks whenever a job has a result to pass back to the module for further processing when it is finished. Whenever the command processor creates a new job it passes the ModuleBase as an INotify-reference parameter to the job so that the job can return results to the ModuleBase in an asynchronous way to its original callers by calling methods defined through the INotify-interface.

A result received through the INotify-interface can result in different things. First, whenever a message-gateway and the module are hosted in process of within a client application, the module can hand the notification received through INotify-methods from jobs back to the message gateway which in turn can throw an event that can be received by other parts of the client application (such as a controller, a presentation model or a view – more on that later). Remember the illustration of in Figure 5 earlier where you can see that the message gateway sits in between of the original caller and the ModuleBase that coordinated the execution of a job.

On the other hand a result received from a job can result in the execution of a further job, as well. This can be a simple job or it can be a job that needs to deliver the result to another service or another client running on a remote machine. With that we need to switch to thinking about how jobs can communicate to other services or clients running on remote machines in general!

Communicating with other Services and Clients

Since the advent of web services and service oriented architecture Microsoft is talking about a pattern when a service-consumer wants to talk to a service-provider on a remote location called the **Agent/Service-Pattern**. The primary idea and motivation of this pattern is de-coupling the consumer-implementation from the service-provider interfaces by encapsulating the logic of mapping internally used objects and data structures to the service provider's data structures and required messages as well as subsequently calling the service. Classes which are encapsulating this kind of logic are called **agents** and each agent is responsible for calling services that are implementing specific service contracts (one agent per service contract).

But agents can do even more! They can deal with online-/offline capabilities such as retrieving data from an offline reference data store whenever the application is not connected to the service. They can implement fallback strategies such as try to call alternate endpoints of a service if the primary or default-endpoint of a service is not available. All that can happen transparently from the remaining parts of the consumer-application (which can, of course, be a client or another backend service).

Frequentis decided to implement a pattern for communication with other service providers that, in my opinion, is a very good example for applying the agent/service-pattern concepts. Their agent/service implementation involves several objects, each defined by a separate interface in the architecture: IConnectionPoint-objects, IClientFacade-objects as well as IRequestFacade-objects.

Connection-point objects are implementing the **IConnectionPoint**-interface and are providing methods for retrieving active connections to services. Each connection to a backend service is implemented in a separate object referred to as **ClientFacadeType**-objects. These objects are responsible for connecting to backend services (e.g. by using WCF-proxies for web services or by using ADO.NET database connections or similar mechanisms). Connection point instances are in a 1:1-relationship with such client façade types and they essentially manage their instantiation based on connection-point-settings as well as their initialization. Reflecting this back to the explanation of the agent/service pattern, the actual agent is implemented in the ClientFacadeType-object while the connection-point decides, when a client façade instance is created (one instance for all requests as a singleton or a new instance for each request by a job) and how it gets initialized by calling its initialize method.

For each service-contract or backend-service type a consumer will have a separate ClientFacadeType implementation. Referring back to Figure 2 that means, that for each system interface (NMEA gateway, DB gateway etc.) a separate ClientFacadeType implementation exists. This system interface is used to talk to the message bus which then forwards requests to responsible backend systems implementing the system interface. All these instances implement the **IClientFacade** interface that forces the implementations to have an Initialize-method. This initialize-method is used by the connection point to initialize the ClientFacadeType instance appropriately. Furthermore each ClientFacadeType implementation implements an interface that itself is inherited by **IRequestFacade**. This inherited interfaces (call them **subsystem-facade**) implement the specific operations a job can call for subsequently calling backend services. Each operation on its own performs the actual service call.

Jobs typically request a ClientFacadeType instance through IConnectionPoint-instances. Whenever that happens, the IConnectionPoint-instance determines whether it needs to create and initialize its ClientFacadeType-implementation, does so if necessary and returns the newly created instance to the job which in turn uses it for calling backend services. Depending on the settings of an IConnectionPoint-instance (singleton or prototype) it creates a new instance of the ClientFacadeType it manages or keeps a single instance for all requests as a singleton.

The interesting aspect of this architecture is, that jobs which are intended to implement business logic and business logic, only, do not need to deal with instantiation-behaviors or calls to other remote services. Jobs therefore can focus on their business logic implementation without the need of dealing with communication-issues or connection-issues to backend-systems. This logic is encapsulated into connection points and client façade type implementations and therefore is kept away from the business logic of the application.

Furthermore interesting is the fact that calling other services is implemented as part of jobs within the whole asynchronous architecture, as well. That means a call to a remote service is a job as any other job that just encapsulates ordinary business logic. It therefore is executed within the worker queue in a separate thread and it does not influence the responsiveness of the application as a normal job. Combined with capabilities of WCF such as one-way calls or MSMQ-bindings this enables a really fail-safe, always-responsive, reliable and also very scalable architecture for the TnT-scenario.

Decisions Specifically Tailored to WPF

The architectural approaches and decisions we've covered so far are general for the overall TnT-system and can be applied anywhere – is it in the client applications running on the working positions or is it in services running on the backend. Within the next few chapters of this paper we will discuss some aspects which are rather specific to the way Frequentis is using **Windows Presentation Foundation** in their application.

All the patterns and approaches you've been reading about in the previous chapters apply to this part as well. Referring back to Figure 5 we will now move to the consumer-parts of message gateways and discuss architectural decisions and their motivations for these parts – just in the context of WPF client applications running on working positions.

Motivation for leveraging WPF

Before we dig into any solution patterns on how Frequentis leverages WPF as a client framework, I think it's interesting talking about the motivation from Frequentis for using WPF instead of Windows Forms as technology for implementing all further clients within the TnT system (the first clients of the system have been developed in Windows Forms before we started leveraging WPF in winter 2008). These primary motivations of Frequentis for leveraging WPF especially in the TnT environment where the following ones:

- **Alternate user interfaces through Styles.**
  WPF supports strong capabilities for modifying the visual appearance of nearly every UI element by providing alternate styles. That means, if a customer wants to display log-messages in a completely alternative way (instead of, for example, just having flat lists), Frequentis could do that by providing other styles for list-boxes without touching any other part of the existing implementation.
- **Productivity and Separation of Concerns through WPF data binding**
  The data binding capabilities of WPF for binding custom object trees to XAML-user interfaces enable Frequentis to implement as much of the UI logic outside of XAML and code-besides of XAML as possible. This helps to dramatically increase testability of UI-based logic and components – even logic that is closer to the actual user interface and XAML.
- **Intuitive, new and easy-to-learn user interfaces**
  Of course that's a classic motivation for using WPF in line-of-business applications and I guess you've heard that one very often without concretely expressing what this means. For this project and for the TnT application it means that Frequentis wants to provide modern user interfaces that are able to visualize and process a large amount of data in a way people are used to from times when they still captured log information in a hand-written fashion. That means the application should look like and feel like having a log-book in their hands but with the ability to capture, view and process a much larger amount of data. Note that this is a long-term future that is not part of the first version of the application, but we see the potential of getting there with WPF simply by using different styles and templates for controls as mentioned in the first argument on alternative user interfaces for different customers.

Below you'll see two different versions of the application screen shots for the TnT logbook applications. The TnT Logbook application is implemented using Windows Presentation Foundation (WPF) of .NET 3.5 SP1. Interesting with these two versions of the screens is that they are exactly the same screens with the same code underneath it. The different visual appearance is completely implemented using XAML-styles in WPF. The development of the new style happened without even touching any part of the code of the application and the XAML-definitions of the windows themselves – the styles are fully outsourced into XAML resource files.

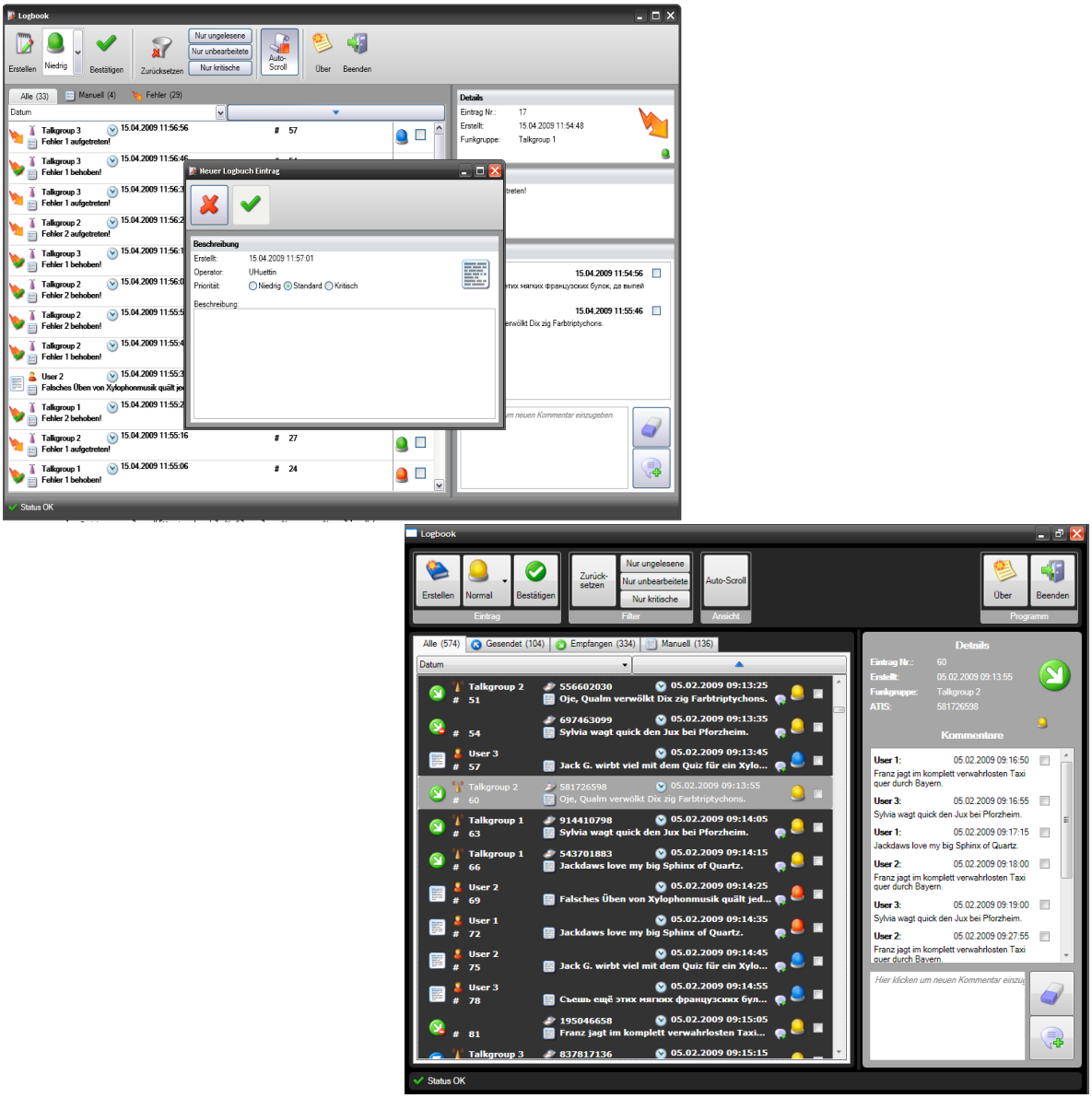


Figure 6: TnT Logbook as the first client implemented in WPF for the TnT working position clients

Note that the application leverages a similar concept as a toolbar as Microsoft Office does with the ribbon control. The main list in the left center of the application shows incoming and outgoing log messages captures by MCS and NMEA and other TnT systems and published to backend services and other working position. Every working position, every ship and any other participant for managing ship vessel traffic is able to consume and produce such messages including comments, statements, categories and severity-levels attached to them. These messages are used for managing both, the ordinary ship vessel traffic as well as dealing with exceptional situations and emergency cases.

The main-list is nothing else than a WPF list box control with a special item template applied to it. Other visualizations would simply involve other item templates and layout templates for the list to support different styles for different customers. Frequentis designed several possible styles for the TnT logbook in other areas (not for real-time monitoring) as below which haven't been implemented, yet but will be evaluated for future versions.

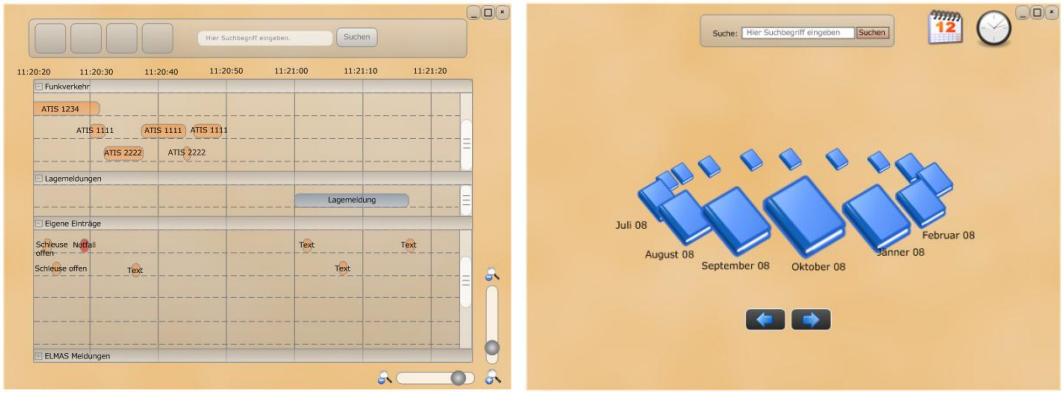


Figure 7: Other user interface designs for the TnT Logbook application

Please also note that the data displayed in the application is just test-data with dummy-test messages – we had to remove productive data from screen shorts for security and privacy reasons!

As you can see, versatile user interfaces and user interface parts of the various applications in the TnT-environment can be implemented very well using WPF with its possibilities for customizing control templates and styles. Of course the different visualizations are not suitable for all areas of the application (the right one definitely not for "critical" parts with the need of "fast" navigation), but they are practicable for some parts of the application. E.g. the time-line view in the left side of Figure 7 is definitely a very clear illustration of what's going on when with the ships.

Presentation-Model and Data binding

To be able to switch user interface styles and designs without modifying any parts of the implementation of the client-side a clear separation between the actual views, the data that is displayed and its client-side processing-logic is necessary. For this purpose Frequentis decided to not have any logic in the code-beside file of WPF views implemented with XAML and C# at all. Even button event handlers or similar code parts should not be part of the code-beside of a WPF XAML-view if possible.

Finally that means all the logic as well as the data that is displayed in views needs to reside in separate classes and components so that XAML views can be replaced without modifying the remaining parts of the application at all. This is where the presentation model pattern becomes important. Typically you would follow the MVC (model-view-controller) or MVP (model-view-presenter) patterns to achieve this goal! With these patterns you would have most of the code in your controller and the controller would bind models (the data to be displayed) onto the XAML-views. But at the end of the day these patterns wouldn't help you completely avoiding code in code-beside files of XAML views. Still you would need some code for catching control-events or adopting a window-title property or similar things.

The presentation model pattern (also known as Model-View-ViewModel pattern) provides a possible solution for better dealing with this issue and therefore Frequentis decided to apply this pattern for all client applications developed with WPF starting with the first implementation in the TnT Logbook application. Essentially this pattern works as follows:

- You still work with models as the data to be displayed and modified by your UI.
- Optionally you still can have a controller or a presenter responsible for coordinating views, preparing models and talking with other client-components or components talking to services (for Frequentis – message gateways on the client to perform job executions).
- But instead having controllers or presenters binding your models directly to views, you create separate classes that contain the model and its properties as well as additional properties which are relevant for the UI and therefore can be used for binding as well. Such additional properties can be a property for a window title or a property for binding a command-delegate to a button-command.

With this approach you can have most of the code that you originally would have in a code-beside in the presentation model. This presentation model can then be bound to different XAML-views and the views are updated through data binding and through data binding, only, without any further ways. That means even button-click events wouldn't be necessary anymore as WPF natively supports the command pattern and therefore allows a button's command-property to be bound to a property of the presentation model that on its own represents a .NET delegate-method for executing the command. The following figure illustrates that architecture:

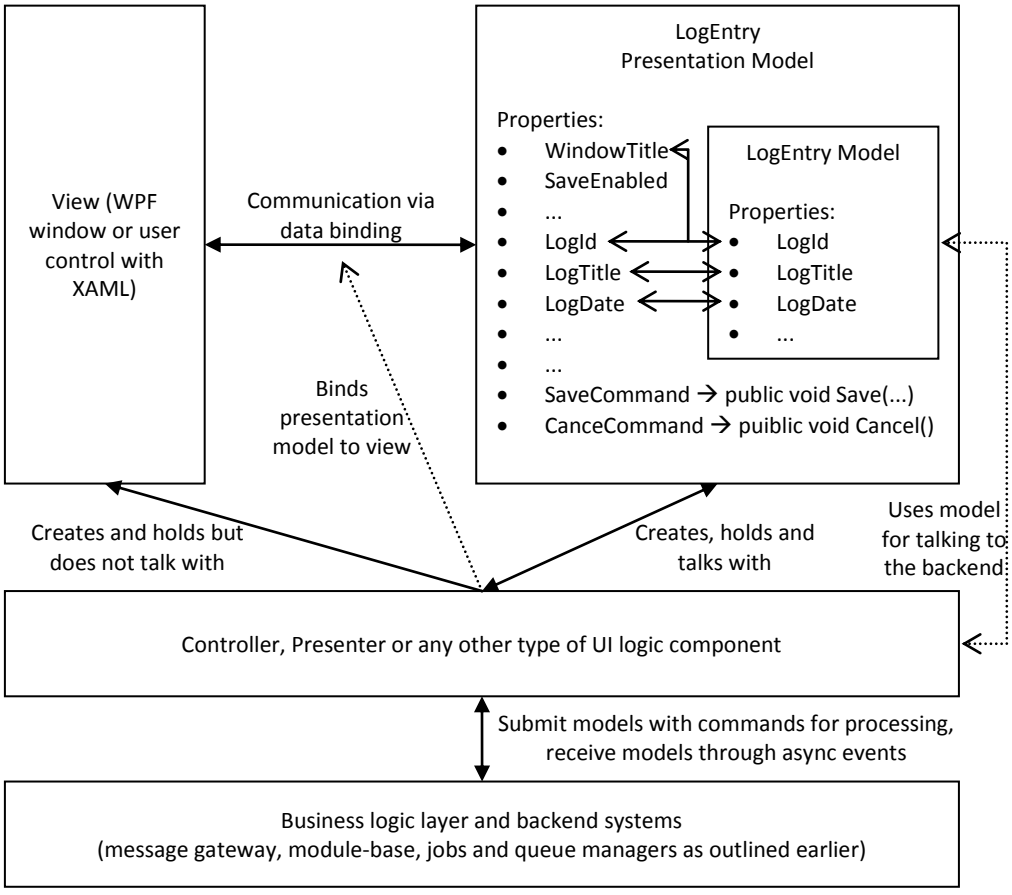


Figure 8: The Presentation Model pattern conceptually illustrated

All properties of the presentation model are then used to be bound to parts of the UI such as the SaveCommand-property which is a reference to a delegate for executing a button-click command or the WindowTitle-property which is used to be bound to the Text-property of the XAML-window. All these properties are updated by the XAML-view automatically through two-way data binding! Whenever that happens and therefore a property of the presentation model gets updated, the presentation model itself can do anything it wants to or needs to do with that change (e.g. by capturing the OnPropertyChanged event of the INotifyPropertyChanged-interface or by directly adding code to its property-setter methods).

Please note that the core of the presentation model pattern is having a separate model that wraps (or decorates) the actual model. You bind this presentation model class to the view instead of binding the model, directly. That's the core of the pattern, having a controller, presenter or something else is an optional aspect. In our case we have a controller kind-of class that is responsible for coordinating the asynchronous communication with the business logic layer (within or outside of the client application).

Also note, that unfortunately there are some limitations when using data binding with WPF in the .NET Framework 3.x (incl. 3.5 service pack 1). You cannot bind events to properties. That means, any "event" that is not available as a command or as an update of a data-property from your underlying data model (such as a list-box' selection-change event) you still need to have code in your XAML's code-beside file to catch this event and forward it to your presenter, controller or presentation model. But, fortunately this limitation will be fixed with .NET Framework 4.0 and the new version of the XAML-specification that will ship with this version of the framework – XAML 4.0 will allow you to bind events to properties of underlying models that do express event delegates!

Okay, now that we know, how Frequentis is leveraging data binding for WPF to keep testability of client code simple and independent from XAML-views with their styles, templates, controls and structures, we need to go one step further and take a look at how an action in the UI such as a button click or a change of a value in a model leads to the asynchronous execution of a job as outlined in the previous chapters!

Connecting the Front-End with the Business-Logic Layer

Now that we now, how the architecture looks like at the very front-end part of the application as well as on the execution of business logic in the business-layer it's time to talk about the glue between those two parts. Essentially you know anything you need to know to understand, how the two things are glued together. We will take a look at two different sequence diagrams for doing so!

In the first sequence diagram we will take a look how an event in the view that ultimately results in a change of the presentation model as well as the underlying model will result into a request that is sent to a message gateway. Please note that I will skip some implementation-specific aspects from this diagram which are not relevant for understanding the architecture and concepts discussed in this paper!

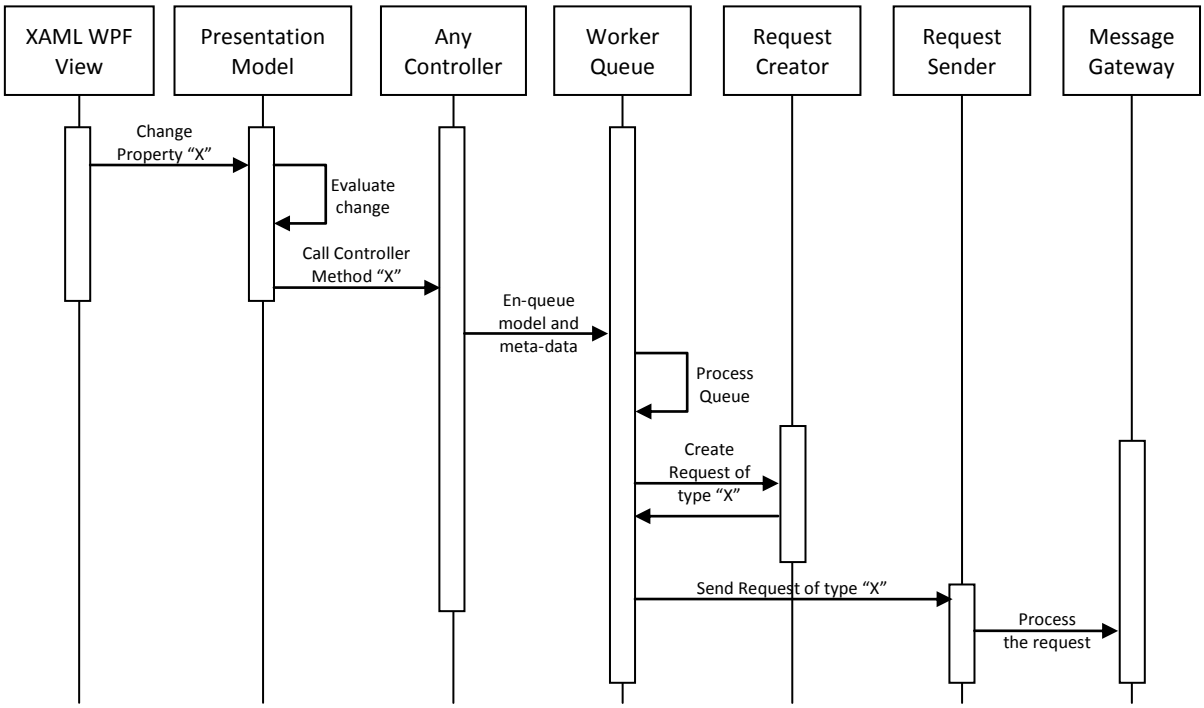


Figure 9: How a change in the UI will lead to the execution of jobs in the business layer

The message gateway itself processes incoming requests as outlined earlier (see Figure 5). Note that each message gateway also supports an event that is fired whenever a job that executes business layer asynchronously notifies the module base that is called by the message gateway through methods defined in the INotify-interface which is implemented by all module base classes! In that case the module base then tells the message gateway to fire its response-event so that callers of the message gateway can be notified asynchronously through events (if they're running in-process, otherwise a job calling other systems through the message-bus must be created).

A very interesting aspect here is that the client hosts worker queues as explained in the previous chapters, as well. These are responsible for asynchronously creating requests to be sent to a message gateway of the business layer that lead to the execution of business logic jobs. With this queue-concept also implemented on the client (through controller classes hosting queues which are executed in their own threads) the user interface is always responsive and is never blocking.

For clarifying and repeating the asynchronous execution of jobs in the business layer we need to take a look at the following sequence diagram which outlines the concepts from Figure 5 at a more detailed level! Note that the ModuleBase is instantiated and called by the message-gateway. Also note that I've skipped some details which are not important for understanding the concepts and which are rather specific to Frequentis' implementation.

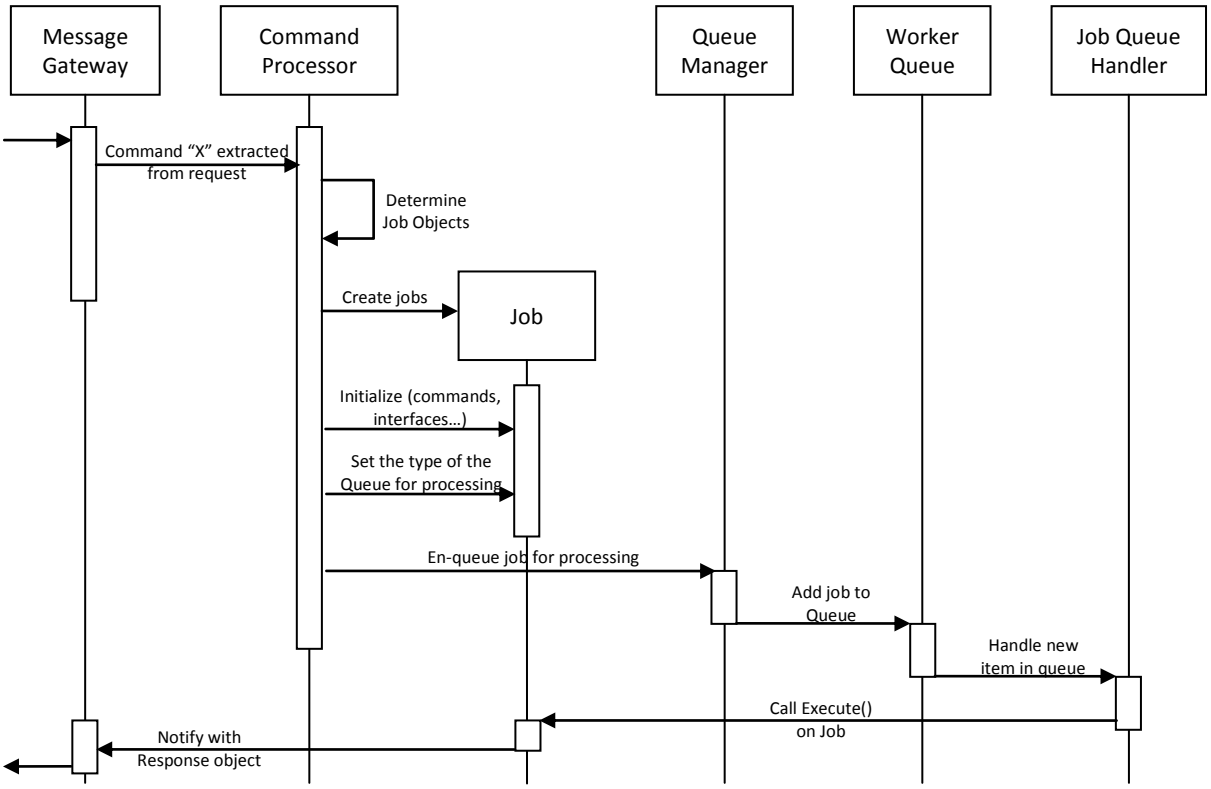


Figure 10: Message flow from the message gateway to jobs and their execution

Please note that every client will host its own business layer as outlined in Figure 5 and therefore the concepts below are valid for every client and of course for backend services as well. If a client needs to communicate to backend-systems, that happens by en-queuing jobs through its own business layer that will leverage connection points for calling remote services. If the whole TnT-framework and the message gateway are hosted in a backend-system, then the gateway will be called by a WCF-service façade implementation for asynchronous job execution.

Communication back to the caller (e.g. the controller originally en-queued an item forwarded to the message gateway as shown in Figure 9) will happen if a job notifies the module-base by calling the Notify-method defined in the INotify interface. The module-base will then forward that notification to the message gateway which in turn throws an event.

This event can be processed by the controller, directly, or it can be processed by the original request sender object (see Figure 9) that puts an item into the client's worker queue which will be processed as any other item as well. Such an item added to the client's worker queue can result in method calls on the controller who then updates the presentation model. Remember that the message gateway way implemented as a class inherited from **ModuleBase**. Thinking about this, of course each different implementation of ModuleBase can decide, whatever needs to be done with a notification from a lower level (forwarded to a caller, create a new job, whatever!). If the ModuleBase-implementation decides to notify the caller through an event, the presentation model in a client application might be the one who is receiving this event. As the presentation model is bound to the XAML-view, these changes will reflect immediately in the UI (given the fact that the presentation model class implements the .NET interface INotifyPropertyChange so that the XAML-view will be able to catch events on property updates).

With that we have taken a look at a complete roundtrip, from asynchronously putting changes of our model as a result from UI-events into a client-queue that results into the creation of requests for message-gateways which are the glue to an asynchronous execution of jobs within the business layer!

Conclusion and Summary

In this whitepaper you have seen some architectural approaches from a real-world project for implementing always responsive, reliable and scalable smart client applications and backend-services based on the ideas and concepts Frequentis implemented in their Tracking & Tracing (TnT) application landscape.

Applications of the TnT landscape are responsible for tracking and monitoring ship vessel traffic in large ports/havens as well as collaboratively working on emergency cases. Applications and services in this landscape need to be always responsive, especially when it comes to dealing with exceptional cases. Failure of one system may not influence operations of any other system at all. To me, the most important decisions for supporting such an application landscape were the following:

- **Asynchronous processing everywhere.**
  Whether it's on the client or on servers, any action results into an item put into queues which are executed by asynchronous worker threads.
- **Job- and command-patterns for defining "asynchronously" executed items.**
  Asynchronous execution in multiple threads dramatically increases the complexity of applications. Therefore you need paradigms and metaphors to be able to better understand what's going on in such systems. Splitting business logic into autonomous jobs or tasks that do not have any relationship to other jobs/tasks is a core pattern evolving in parallel processing. Frequentis applied this pattern and approach in their application architecture!
- **Peer-2-Peer message-bus architecture with WCF.**
  I very much liked the idea of a peer-2-peer message-bus for ensuring a reliable communication with the least possible effort. Peer-2-peer based communication happens between all the nodes connected to the same peer-cloud without having some kind of central servers in the game. If one node fails, the others aren't affected by this failure because they're talking to each other node, directly. Unfortunately Frequentis determined that in its current implementation the peer-2-peer protocols on Windows do not scale after a certain load stresses the system. Therefore they will provide an alternative deployment architecture where they will need to establish a load-balanced cluster of WCF-services where actors can subscribe as publishers and subscribers to messages within the TnT landscape. The effects on existing service implementation should be minimal as WCF keeps protocol and communication details away from the core logic of the application.
- **Intuitive user interfaces with different styles per customer**
  Also I liked the idea of how Frequentis leverages WPF with its control templates and styles for supporting different visualizations of large amounts of data in a for end users effective, easy to handle way. The implementation of MVC/MVP or presentation model patterns helps Frequentis building UI-logic that is easy to test and independent from visualizations built for different customers using XAML and WPF.

I expect to see more and more architectures of this kind with the increasing importance of parallel computing as well as technologies for parallel computing coming up on the market. Taking a look at them clearly shows the advent of a task/job-based paradigm for really scalable and always responsive applications that will benefit from multi-core and multi-processor machines.

With the knowledge from this paper and project in your mind, you are definitely equipped for the new world of development of parallel and scalable applications.