



ASSIGNMENT OF MASTER'S THESIS

Title: Monitoring proprietary .NET processes on client's servers
Student: Bc. Ondřej Janáček
Supervisor: Tomáš Nouza, M.Sc.
Study Programme: Informatics
Study Branch: Web and Software Engineering
Department: Department of Software Engineering
Validity: Until the end of summer semester 2016/17

Instructions

The objective of the thesis is to design and implement a system, built on the .NET platform, that will monitor customer's servers state and mainly proprietary .NET processes running there, collect data (metrics) about and from certain processes, allow to set up actions to be executed when certain conditions are met, and display the collected metrics to users.

The system requirements:

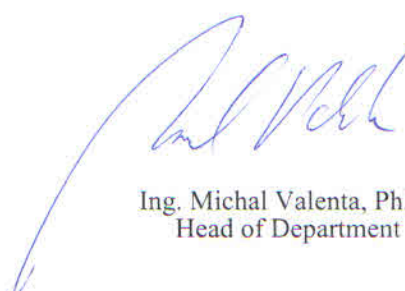
- It consists of a server (sp) and a client desktop application (cp).
- The sp monitors and collects specified metrics and sends them to cp when requested.
- The sp is able to analyze metrics and react to certain values according to preset rules.
- The cp displays the current state of monitored servers and processes.

Thesis tasks:

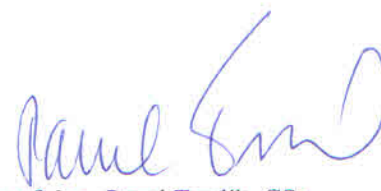
- In cooperation with the supervisor, specify metrics to be monitored and collected by the sp and rules that allow to specify the sp reaction to certain metrics values.
- Design and implement the system.
- Test and document the whole system.

References

Will be provided by the supervisor.


Ing. Michal Valenta, Ph.D.
Head of Department




prof. Ing. Pavel Tvrdič, CSc.
Dean

Prague November 19, 2015

CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF SOFTWARE ENGINEERING



Master's thesis

Monitoring proprietary .NET processes on client's servers

Bc. Ondřej Janáček

Supervisor: Tomáš Nouza, M. Sc.

9th May 2016

Acknowledgements

I would like to thank to both my supervisor Tomáš Nouza, M. Sc. and my reviewer Ing. Miroslav Prágl, MBA for all the valuable tips and recommendations.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on 9th May 2016

.....

Czech Technical University in Prague
Faculty of Information Technology

© 2016 Ondřej Janáček. All rights reserved.

This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Janáček, Ondřej. *Monitoring proprietary .NET processes on client's servers*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2016.

Abstrakt

Tato práce popisuje návrh a implementaci vlastního monitorovacího systému, primárně určenému k monitorování procesů, běžících na platformě .NET Framework, z venku i ze vnitř. Výstupem je funkční systém složený z WPF klienta, reportovací knihovny a monitorovací služby.

Klíčová slova monitorování, .NET, proces, reportování, metriky

Abstract

This thesis describes design and implementation of a custom software stack for monitoring of processes, running on .NET Framework, both from outside and inside. The output is a working monitoring system composed from a desktop WPF client, a reporting library and a monitoring service.

Keywords monitoring, .NET, process, reporting, metrics

Contents

Introduction	1
Motivation	1
1 Analysis	3
1.1 Target	3
1.2 Functional requirements	4
1.3 Non-functional requirements	7
1.4 Use case scenarios	8
1.5 Existing solutions	9
1.6 Conclusion	12
2 Design	13
2.1 Architecture	13
2.2 Metric system	17
2.3 Client user interface	18
2.4 Conclusion	22
3 Implementation	23
3.1 Technologies used	23
3.2 Architecture	28
3.3 Metric system	29
3.4 Server	29
3.5 Client	33
3.6 Points of interest	35
3.7 Conclusion	37
4 Testing	39
4.1 Unit tests	39
4.2 Integration tests	40
4.3 Performance testing	40

4.4	Monitoring client UI	43
4.5	Conclusion	48
Conclusion		49
	Moving forward	50
	What's next	50
Bibliography		53
A Acronyms		55
B Installation guide		57
C User guide		59
	C.1 Reporting module	59
	C.2 Monitoring client	59
D Contents of enclosed DVD		61

List of Figures

0.1	Logis CAD solution	2
1.1	Simplified deployment model	4
2.1	Architectural overview	14
2.2	MVVM components relationships	17
2.3	Dashboard design - the first iteration	19
2.4	Dashboard design - the second iteration	20
2.5	Server view design	21
2.6	Process details view design	22
3.1	Monitoring System modules	28
3.2	Metric types hierarchy	29
3.3	Fraction metric appearance	33
3.4	Dashboard implemented	34
3.5	Metric history	35
4.1	Code Lens	40
4.2	Integration test	41
4.3	Client performance tracing	42
4.4	Client memory profiling	42
4.5	Problematic resource	44
4.6	Problematic metric	44
4.7	Ongoing long-running task	45
4.8	Finished long-running task	45
4.9	Server view window title	47
4.10	Client error	48
4.11	Dashboard future	51
B.1	Installation guide result	58

Introduction

Monitoring in computer science generally means using some sort of a measuring unit in order to be aware of the state of a system by observing it for any changes occurring over time [1]. This measuring unit is collecting and storing state data, sometimes called *metrics*. However, only collecting and storing would not be much useful without having means to report and interpret these metrics to users of the system. And so there's usually a second piece of software that understands the collected data and displays it to users in a way that allows quick and easy reasoning and decision-making.

There are many things about computer systems that ask for being monitored. System resources, websites, network and computer users, just to name a few examples. We can observe for events, performance characteristics or level of resources. The problem can be approached actively by polling data or lazily by accepting data pushed in our direction or even by combination of both.

Motivation

Logis Solutions offers Computer Aided Dispatch (CAD) solutions to the Emergency Medical Services market. Their product is categorized as Software as a Service and is deployed to a customer's server infrastructure. It's the customer who takes responsibility for stability, maintenance and monitoring of the hardware and Operating System (OS). Picture 0.1 shows a few screenshots from the product.

However, it's also crucial that OS processes running under the hood of the product are closely monitored to spot and prevent any critical errors that could lead to unavailability of some parts or the whole system. There's only so much that one can see just by observing how much system resources a process consumes. It's not easy to reason about a process behavior without a deeper knowledge of its internals. The goal of this thesis is to fill this gap

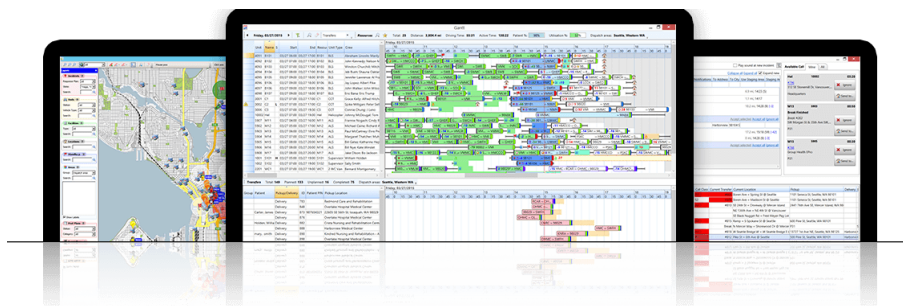


Figure 0.1: Logis CAD solution screens

by providing tools to developers and a support team to describe expected behavior and observe for any deviations.

This document is structured into chapters based on major phases of software development life cycle. It starts with analysis – defining target, specifying requirements and existing solutions study. Follows design, laying out architecture ideas and options before the implementation part takes over focusing on the specific technologies and inner logic. Then there's testing focusing on the code quality and real user experience, and it's all finished with a conclusion bringing new ideas for future development.

Analysis

The whole product is built and runs on the full .NET Framework and therefore requires some Windows OS running underneath. It's usually some version of Windows Server on customer's servers and test machines. Most of the product modules run as Windows service applications – long-running executable applications that run in their own Windows sessions [2], without any User Interface (UI). It's these services that need to be monitored. Metrics differ a service from service and more often than not, it's only an author of a service who knows what are the most important information inside it worth monitoring. In the scope of this document, I refer to instances of these services as .NET processes, because technically a Windows service built on .NET Framework spawns a process, which relies on .NET Framework during its run-time, when started.

This chapter describes target use case scenarios for the system, specifies functional and non-functional requirements that should make all the use cases possible, and is concluded by a research into existing solutions.

1.1 Target

The output of this thesis should be a software stack that, when properly configured, forms a working monitoring system. It should consist from the following three main software modules (depicted in diagram 1.1):

- A module used within .NET processes to report metrics. It should provide means to both receive requests for metrics and send them, without being requested, to a predefined endpoint.
- A server part – a module, which serves as a mediator between clients and a machine it's running on, responsible for collecting metrics from processes and the machine's OS.

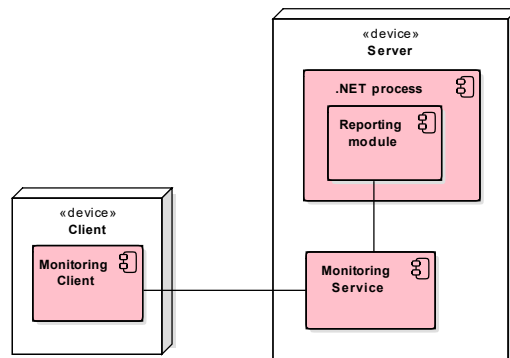


Figure 1.1: The three main components of a monitoring system

- A client part – a Graphical User Interface (GUI) application displaying collected metrics received from a server in a human-readable and easily understandable format.

The functional and non-functional requirements in the following sections are categorized by this separation. Furthermore, it's necessary to

- come up with a format to describe all possible metrics without knowing what exactly they are (some are known and listed in the next section) and
- introduce a simple commanding solution in order to be able to communicate with .NET processes from the client part of the system. Although this is not a feature related to monitoring at first sight, it's something that monitoring in person leads to in some cases described in section Use case scenarios and would be a tremendous help.

1.2 Functional requirements

Follows a list of all the functional requirements divided by the modules and furthermore formed into groups representing the module's main features.

1.2.1 Reporting for .NET processes (services)

- Reporting
 - Decide whether to push metrics or send upon a request.
 - Send a package of metrics to a predefined endpoint.
- Commands

- Define commands in code and make them available through an interface.
- Receive a command from outside the process and return a response.

1.2.2 Server part

- Monitoring
 - Receive metrics from .NET processes actively pushing metrics.
 - Poll registered .NET processes that accept metric requests, and be able to detect a new process and connect to it.
 - Configure monitoring targets and data sources along with polling frequency:
 - * Random Access Memory (RAM),
 - * disk drives,
 - * Windows event logs,
 - * processes,
 - * Microsoft SQL databases,
 - * .NET processes using the polling model.
 - Store the latest metrics from all monitored targets.
- Reporting
 - Receive a connection from a client.
 - Send the latest metrics to a client when requested.
 - Configure reactions – apart from watching the current state by users, there should be an option to set reactions to specific events.
 - * Reaction types
 - Log a message into Windows event log.
 - Send an email.
 - Restart a process.
 - Composite of any of the previous reactions.
 - * Reaction triggers
 - Amount of RAM used by OS exceeds a threshold.
 - Amount of disk space left on a certain disk drive is below a tolerable value.
 - Certain event ID appears in a Windows event log, once or repeatedly within a time span.
 - Amount of RAM or number of processor cores used by a process exceeds a threshold.

- Process exits.
- Database size or database log size exceeds a threshold.
- Commands
 - Carry out a command sent from a client and eventually return a result.
 - * Create a process memory dump and return a message with information.
 - * Query a Windows event log and return event log entries.
 - * Remove exited process from monitoring.
 - Forward a command to a .NET process and return a response from the process if there's any.
- Configuration
 - IP address and port where requests from clients are accepted.
 - Flexible logging configuration.
 - Destination path for storing finished memory dumps.
 - Email sending related settings.

1.2.3 Client part

- A single user role – the project owner's support team.
- Monitoring
 - Configure:
 - * network timeouts,
 - * servers to connect to,
 - * *environments* – in the context of this thesis, an environment is a group of machines in the same network that distribute load by running demanding processes of the product on different machines.
 - Establish a connection to a server and request the latest metrics.
 - Show all configured servers grouped into environments.
 - Show details for a server – all metrics received from the server.
 - * When a process stops, keep it shown in the details with a warning and allow to remove it to fix the issue.
- Inform a user about important information through a log window and allow to save and clear messages.

- Instruct a server to
 - create a process memory dump,
 - send a part of a Windows event log,
 - forward a command to a monitored .NET process.

1.3 Non-functional requirements

The project owner requires a lightweight monitoring system that's easily configurable for different customer environments, is easy to set up during initial deployments and offers extensibility and customization in order to work well with internal tools. Being lightweight in this case means not doing anything above what's required and collect only relevant data. Easy configuration and customization is essential as every customer can use different number of different CAD modules.

For both the server and client part, configuration should be stored in eXtensible Markup Language (XML) files in order to leverage configuration transformations during build on the project owner's build server. Configuration change during run-time is not required as the correct configuration is stored in a Version Control System (VCS) and all changes should originate from there. Also correctness of configuration is verified in a preproduction environment.

1.3.1 Reporting for .NET processes (services)

- Preferably implemented as a .NET class library or a .NET wrapper (a .NET library that calls into a component not built on .NET Framework) for easy and straightforward usage from .NET services.
- Sending metrics out and receiving commands must not interfere with services' business logic.
- Commanding channel in a process should not be directly accessible from outside of the machine where the process is running.

1.3.2 Server part

- Able to serve more (couple of) clients at the same time, but scalability is not required because of limited number of concurrent users.
- Communication between components does not have to be encrypted.
- Ability to reconnect to a monitored service that stopped and started again later.
- Runs on Windows.

1.3.3 Client part

- Capable of communication with around a dozen of servers simultaneously.
- Ability to renew connection to a server that became unavailable for a time.
- English user interface localization.
- Runs on Windows 7 and higher.

1.4 Use case scenarios

Usually, there are specified user roles involved in a scenario, but there's no mention of users involved in the following scenarios due to having only a single user role.

1.4.1 Regular system check

Steps:

1. Run the client with configuration for a particular customer.
2. See if there's an issue on any of the configured servers or if there was an error logged recently. Follow the Troubleshooting steps if needed.

1.4.2 Post-deployment support

Pretext: A new version of the product has just been deployed to an environment.

Steps:

1. Run the client with configuration for a particular customer.
2. Observe for potential new errors being logged or a server reporting an issue.
3. Should any of the previous happen follow the Troubleshooting steps.

1.4.3 Troubleshooting

Pretext: An issue (a situation that's not expected to happen) was found on a server, the client is already running.

Steps:

1. Show the server details and locate the process or resource causing the issue.

2. If it's a resource causing the issue, it's probably necessary to remotely connect to the server and troubleshoot there. The next steps apply only for processes.
3. Instruct the server to send entries from a Windows event log which the problematic process uses to log events.
4. Filter received entries to display only those related to the process.
5. Based on the event log entries from the previous step decide what's the next best step or their combination in order to solve the issue.
 - Take the process memory dump.
 - Solve the issue by sending appropriate commands to the process.
 - Remotely connect to the server and finish troubleshooting there.

1.4.4 Process interaction

Pretext: A new service has been recently deployed or there were major changes in an existing one.

Steps:

1. Run the client with configuration for a particular customer.
2. Navigate to the server running the service and show its details.
3. Find process running the service and interact with it using commands or check the internal metrics.

1.5 Existing solutions

Before I started searching for existing solutions I narrowed my search by the following considerations.

- Obviously, there's a lot of products targeted at server monitoring but the main data source for the proposed system are process internals. Only basic information about server resources and nothing about user activity on a server or network traffic is required. That's because a customer takes care of these areas. It's good to know, how the product affects the whole system, however.
- Only the latest state of each monitored data source is required to be stored and displayed in the monitoring client. This is because the project owner already has an operational platform for document (in terms of log files) storing, querying and visualization.

- There's probably not a working existing solution because of the very specific requirements. Aim of this search is to find suitable components that could be used in the final system.

1.5.1 ELK stack

So called ELK stack is a composition of Elasticsearch, Logstash and Kibana, all products created by Elastic Inc. It's presented as an end-to-end stack that delivers actionable insights in real time from almost any type of structured and unstructured data source [3], which is mostly what the requirements are. That's why the whole solution is reviewed here. Let's look at them one by one and then as a whole.

1.5.1.1 Elasticsearch

Elasticsearch is a distributed, open source search and analytics engine, designed for horizontal scalability, reliability, and easy management. It combines the speed of search with the power of analytics via a sophisticated, developer-friendly query language covering structured, unstructured, and time-series data. [4]

It is basically a NoSQL document database combined with a powerful search engine and because it's written in Java, it runs on Java Virtual Machine. It has many benefits but there's no need for a database in this project.

1.5.1.2 Logstash

Logstash is a flexible, open source data collection, enrichment, and transportation pipeline. With connectors to common infrastructure for easy integration, Logstash is designed to efficiently process a growing list of log, event, and unstructured data sources for distribution into a variety of outputs, including Elasticsearch. [4]

There's a great number of input and output plugins to transfer data from a data source to data destination. Unfortunately, Logstash only transports data received through an input plugin, it can't poll data sources like system memory or process internals. If I decided to use Logstash as a transportation pipeline, I would still need a component that periodically polls data sources and then pushes received data to a Logstash input plugin, which I see as an extra layer of abstraction just to separate collection from transportation.

1.5.1.3 Kibana

Kibana is an open source data visualization platform that allows you to interact with your data through stunning, powerful graphics. From histograms to geomaps, Kibana brings your data to life with visuals that can be combined

into custom dashboards that help you share insights from your data far and wide. [4]

It's rich visualization capabilities can't be compared to the current plotting solution, but there's no easy way to make Kibana interact with monitored processes through a monitoring service. Its sole purpose is data visualization and discovery.

1.5.1.4 Conclusion

ELK stack is a very fast, powerful and universal platform for working with data. There are more products from Elastic Inc. that can make it even more powerful. Unfortunately, it's not exactly lightweight and lacks many required feature. The good thing is, it's open sourced and could be extended and customized. The bad thing is, it's written mostly in JavaScript and Ruby, their code base is quite extensive and I don't have enough experience with these languages.

1.5.2 Collectl

There are a number of times in which you find yourself needing performance data. These can include benchmarking, monitoring a system's general health or trying to determine what your system was doing at some time in the past. Sometimes you just want to know what the system is doing right now. [5]

A performance monitoring tool. It periodically collects data about system resources and stores them. Unfortunately, it runs only on Linux.

1.5.3 Windows Performance Monitor (PerfMon)

Windows Performance Monitor is a Microsoft Management Console snap-in that provides tools for analyzing system performance. From a single console, you can monitor application and hardware performance in real time, customize what data you want to collect in logs, define thresholds for alerts and automatic actions, generate reports, and view past performance data in a variety of ways. [6]

Performance Monitor satisfies a lot of functional requirements in terms of monitoring, alerting and reactions. It can even connect to a remote machine. However, it can only be connected to a single machine at one moment and I could not possibly describe it as a user friendly tool. And there's no way how to extend it with other required features.

1.5.4 Windows Management Instrumentation

Windows Management Instrumentation (WMI) is the infrastructure for management data and operations on Windows-based operating systems. The WMI

.NET classes enable you to automate administrative tasks and supply information about computer system components such as the version of an operating system, the free space available on a hard drive, the IP address of a computer, or a variety of other data about a computer system. [7]

WMI is one of few available infrastructures for querying system data. In comparison to not-WMI .NET types dedicated to the same task, WMI offers more, but is slower and queries are not statically typed. Fast and reliable code is more important to me, and all specified data source in requirements can be queried without WMI anyway.

1.5.5 Performance Counters

Windows performance counters allow your applications and components to publish, capture, and analyze the performance data that applications, services, and drivers provide. [8]

I considered using performance counters to accomplish two requirements. Firstly, to collect data about system resources like available RAM. They work well enough, but again, .NET provides Application Programming Interface (API) that provide enough functionality, is easy to use and code is so much shorter. Secondly, to report metrics from processes, but it turned out that a performance counter value can only be expressed as a number, which is very limiting. Finally, performance counters are best used together with Performance Monitor, but I already dismissed that idea.

1.6 Conclusion

In this chapter, I specified target of the thesis with all the requirements and use case scenarios. I conducted a research for existing solutions and found out that there are many monitoring related applications, components and even whole software stacks, but none would make my task easier. They either provide too much or too little functionality, or their usage would incur more work than writing my own solution. That's not a bad situation to be in, however. It's always easier to fix an issue in my own code than in third-party solutions.

Design

This chapter presents an architecture for a monitoring system and delves into design decisions of individual system modules. There's also a section dedicated to UI design of the client part.

2.1 Architecture

Architecture is the highest-level breakdown of a system into its parts; the decisions that are hard to change; there are multiple architectures in a system; what is architecturally significant can change over a system's lifetime; and, in the end, architecture boils down to whatever the important stuff is. [9]

Diagram 2.1 shows an architectural overview for the system from the deployment point of view. It displays a monitoring client application running on a client machine, exchanging data over a network with a monitoring service application running on a server machine, which in turn communicates with local .NET processes trying to collect their internal data. A monitoring client connects remotely most of the time. It can also connect from the same device as the service is running on.

2.1.1 Communication protocol

There's a few mechanisms for local interprocess communication supported by Windows. For communication between a monitoring service and monitored processes, a mechanism supporting duplex channel is required to allow the service ask for data and processes to respond. Pipes and network sockets would be both viable solutions, but sockets would require every process listen on a different port. For this reason, I have chosen named pipes because they allow to name a pipe after a process ID, which is always unique.

A pipe is a section of shared memory that processes use for communication. The process that creates a pipe is the pipe server. A process that connects to a pipe is a pipe client. [10] There are two types of pipes:

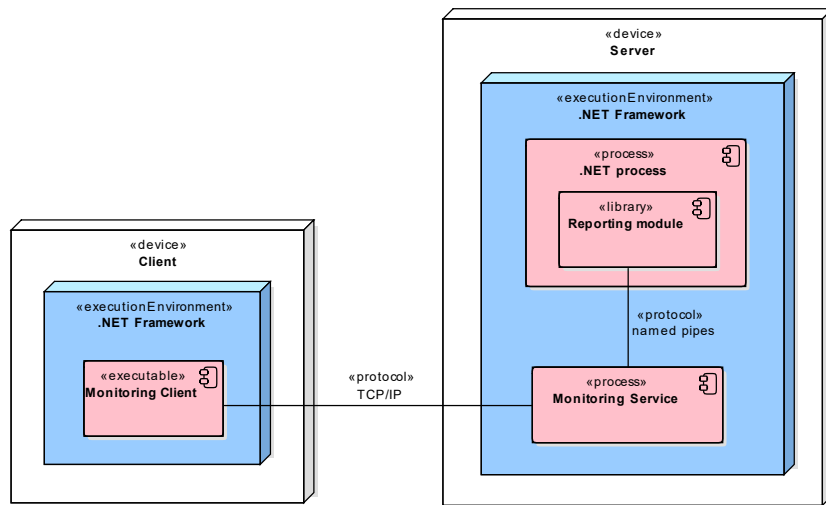


Figure 2.1: Architectural overview

- Anonymous pipes – an anonymous pipe is unnamed, one-way pipe that typically transfers data between a parent process and a child process, and cannot operate over a network.
- Named pipes – a named pipe is a one-way or duplex pipe for communication between the pipe server and one or more pipe clients. All instances of a named pipe share the same pipe name, but each instance has its own buffers and handles, and provides a separate conduit for client/server communication. The use of instances enables multiple pipe clients to use the same named pipe simultaneously. [11]

Although named pipes can be used for communication over network, they fare poorly in bad conditions when compared to sockets. More often than not, a monitoring client is going to communicate with monitoring services over a slow Virtual Private Network (VPN).

2.1.2 Monitoring service

Monitoring service is composed from a few different types of workers – components doing different work. This section describes what these workers do and section Monitoring service in chapter Implementation describes how they do it.

2.1.2.1 Pipe server

A pipe server accepts and stores metrics sent from processes that opted for push model instead of being polled.

2.1.2.2 Socket server

A socket server accepts requests and commands from clients. Based on a request header, it decides what to do with it. Simple requests are processed directly by the server, but those initiating long-running tasks are scheduled for execution by a scheduler. For every request a response is sent immediately. Results of long-running tasks are sent back when available.

2.1.2.3 Scheduler

A scheduler is a worker that accepts work from other components in the system and manages an execution plan. It's an important part of Monitoring service as most of the work being done there are scheduled tasks. It recognizes two main types of tasks.

- Simple task – a single task that's scheduled to be executed as soon as possible or after a period of time (a deferred task). When it fails, it's repeated again after a waiting period.
- Recurring task – a task that is being repeated over and over again with defined periodicity.

An example of a simple task is a process memory dump. A recurring task is checking for new processes that should be monitored.

2.1.2.4 Monitors

A monitor is a component that's responsible for monitoring of a certain system resource. There's a monitor for every kind of a system resource (Process Monitor, Database Monitor, etc.). Based on Monitoring service configuration, each monitor is given information about resources to monitor – *monitor units*. Each monitor only needs to know:

- How to gain data from its units.
- Where to send the data.
- Who to ask to execute a reaction on a situation when it occurs in a unit.

Some monitors also need to know how to find a new unit; Process Monitor, for example, needs to know how to check if there's a new process to follow. Each monitor unit holds the following information.

- How often it should be checked and when was the last time it was checked.
- Current status – data that differ unit type from unit type.
- What situations to watch for.

So for a disk drive, it could be:

- Check every 30 minutes, the last time was 21 minutes ago.
- There's currently 6 GB of free disk space, which means that 26 GB is occupied knowing that the total size is 32 GB.
- Watch for a moment when only 1 GB of free disk space is available.

Each monitor unit type also defines how to create a metrics package from status data and how to evaluate situations. A situation is basically what I call a problem or issue later on – something happened and needs attention – be it low disk space, a stopped process, too much certain events logged, and so on. Some situations happen only once and it's a real problem, whereas some has to happen many times within a time span. Therefore, each situation defines:

- What type of situation it is (examples are already mentioned above).
- How many times it occurred and when it was the first time.
- How often it has to occur within a time span in order to trigger a reaction.
- A reaction that should be executed.

Follows an example from the event log domain.

- An entry with ID 1234 was written to the event log.
- The first one appeared at 01:00 AM on the 12th of May 2016 and another 20 appeared since then in the span of two days.
- If it occurs 50 times in the span of three days,
- send an email to a support team.

When it's all put together, a monitor fetches new data from a data source and updates an appropriate monitor unit. The monitor asks the unit to evaluate its situations and report back all that occurred. Then it asks the unit for a metrics pack. Finally, the metric pack is sent to a central metric store and situation reported from the unit are sent to a situation processor.

2.1.3 Monitoring client

2.1.3.1 Model-View-ViewModel pattern

Model-View-ViewModel (MVVM) is just another design pattern used for building of UI applications. Its main benefits are separation of concerns and testability. Better testability is achieved by separating the view logic from

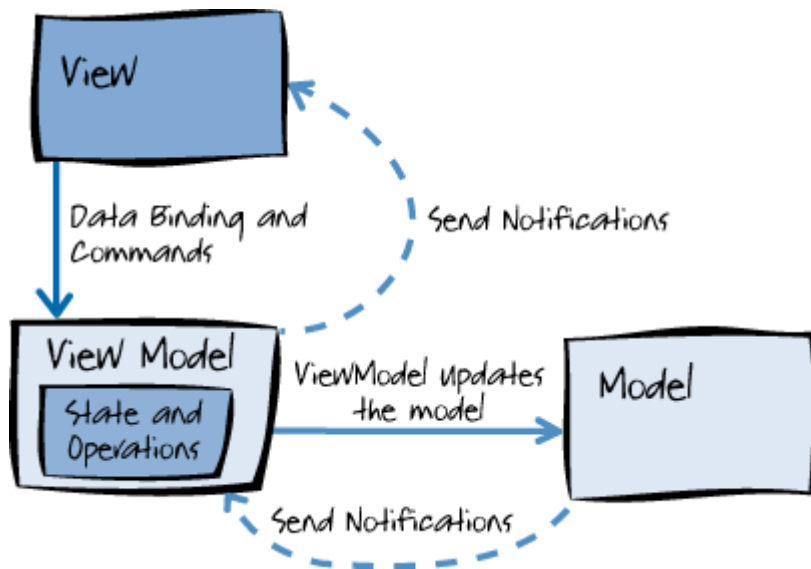


Figure 2.2: Relationships between the MVVM components. Image source: <https://msdn.microsoft.com/library/hh848246.aspx>

business logic (the model), by placing the view model in between. The view model is aware of the model but knows nothing about the view. It communicates its intents to the view by using abstract patterns known to the view. In case of diagram 2.2, the view has a notion of data binding and commands which are partly based on the publisher-subscriber model.

I've chosen MVVM because an application like this has large growth potential and separation of concerns is essential when there's a chance that another client with a limited set of functions will be added to the stack later on.

2.2 Metric system

The idea is to have a set of basic metric types (data types, in other words) and some metric containers that allow composition, in order to be able to describe almost any kind of measurable data. Follows a list of basic metric types with indication of usage and examples.

- Boolean – something happened or is happening. A process is running, for example.
- Numeric metrics:
 - Integer, Double and Long – an amount or degree; most of the time 32-bit integer is enough, but when it's not, 64-bit integer (Long) is used.

- Fraction – a metric composed from two Double metrics to accommodate ratios or percentage in precise numbers instead of percentage that Double could represent. This allows free disk space to be expressed like 3.2 GB / 7.5 GB, for instance.
- Time metrics:
 - TimeSpan – how long something took or lasts.
 - DateTime – a date and time of the last or next occurrence of something.
- Status – a predefined set of common options expressing a state or status of something. Options can list Busy, Error and Connected, for example.
- Text – anything that can't be expressed by any of the previous metric types, like a text message.

Metric container types:

- MetricGroup – a named collection of same or different metric types listed above.
- ProcessMetrics and SystemMetrics – a named collection of metric groups enriched with a source process name in case of ProcessMetrics

This structure does not allow insert groups of metrics in a group to prevent recursive structures. My supervisor agreed that flat structures should cover most of scenarios and in those rare cases when a deeper structure would make sense, more flat ones can be created and structure can be achieved by naming conventions like namespaces in .NET Framework.

2.3 Client user interface

This section contains user interface description and sketches for the client part which consists of two main windows. Monitoring client is the only part of the monitoring system that needs a rich GUI.

2.3.1 Dashboard

Dashboard is the first windows users get to see after the client starts. Most of the screen is occupied by server nodes grouped into environments 2.3. Just by looking at it should be clear if there are any issues on any of the servers. If there are more nodes that a screen can display a scroll bar appears, but otherwise stays hidden.

There's also a log panel with important information for users snapped to a side. Messages of different importance could be differentiated by colors to easily distinguish information from warnings and errors.

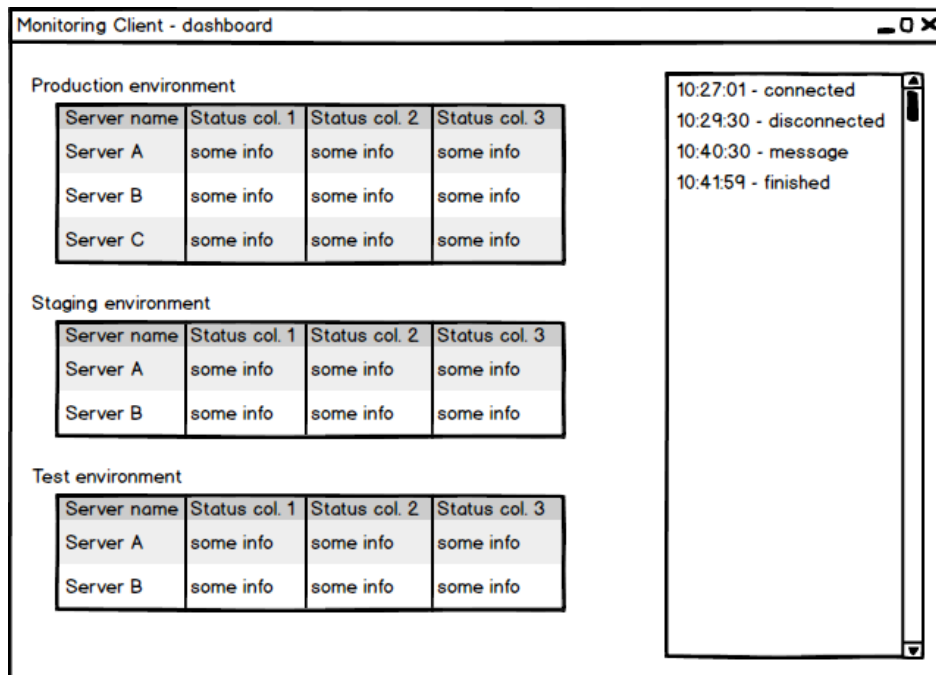


Figure 2.3: The first iteration of Dashboard design

In the first iteration I decided to represent an environment by a table and a server node by a table row. Each piece of a server status information is a table column. See sketch 2.3.

Advantages:

- No wasted space, fits a lot of nodes on a single "page". If a table component that supports grouping was used, there could be only a single table where each environment would be a group. That would save even more space.
- Most of the table components support both column and row ordering, and some support filtering.

Disadvantages:

- With more status information for each server node there would be more columns which would result in horizontal scroll bars in all tables and some columns always hidden.
- Tables are hard to read when there's too much information in them. They have very dense structure.

The longer the system will be used, the more information deemed as important will be required to show in server nodes. For this reason, I decided

2. DESIGN

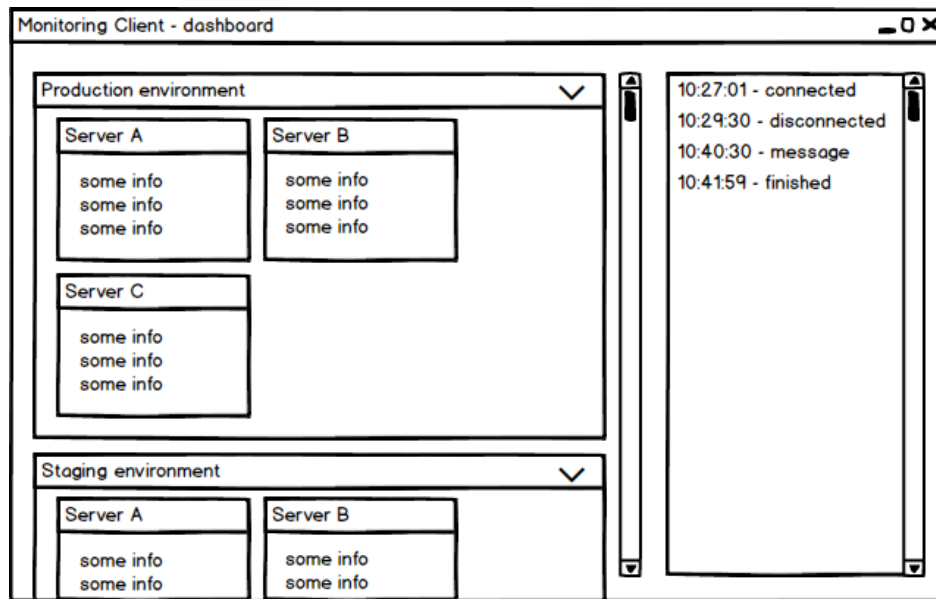


Figure 2.4: The second iteration of Dashboard design

on a different design, based on tiles and wrapping, illustrated in sketch 2.4. Environments are represented by expandable containers, server nodes by tiles and each piece of a server status information is a text line (possibly with an icon to emphasize the meaning).

Advantages:

- Information in a server node stack vertically. All information for a server node are visible when the server node is visible. In case there's a lot of information for each node and lot of nodes, only a single scrollbar is required for browsing as opposed to both horizontal and vertical in the previous design.
- Tile design became popular with newer versions of Windows. It helps to reduce density of information displayed.
- A whole environment can be collapsed

Disadvantages:

- With wrapping, more space is wasted, but can be optimized by resizing the window.
- No sorting and ordering options of out the box.

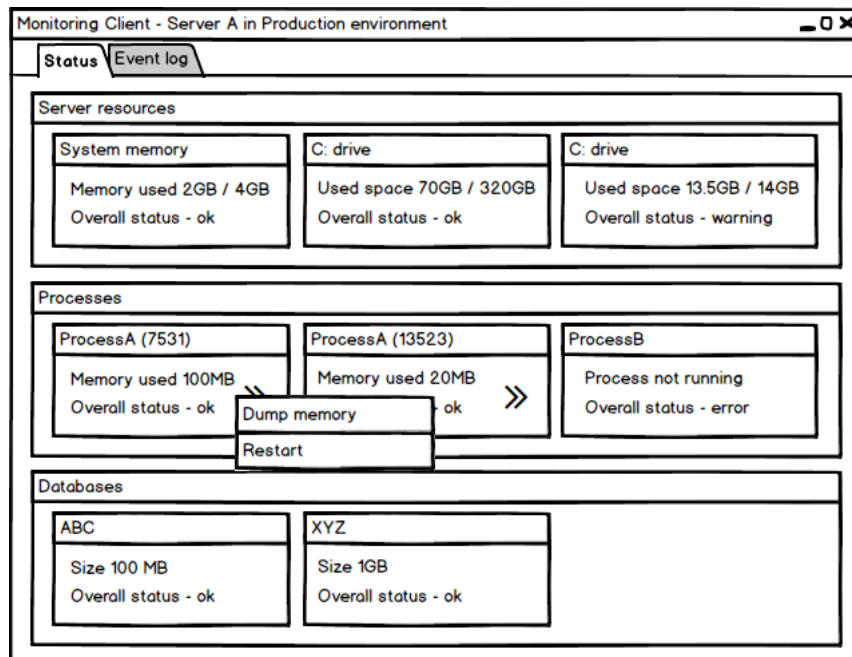


Figure 2.5: A server view

In the end, it depends on user preferences, but a good solution might be to include both layouts and let a user switch in between them. In any case, it would make sense to include a summary at the top, showing a number of problematic server nodes and their names with a quick navigation option. This would help when a number of environments or server nodes is large. I'll consider these features for the next iteration.

2.3.2 Server view

A server view opens upon clicking a server node's title in the dashboard window. Here a user can see all metrics reported from the server, periodically refreshed. If all of them don't fit, scrollbars appear where needed, but otherwise stay hidden.

To reduce the amount of information in one view, some functionality could be available on a different tab or in a pop-up window that appears when requested by clicking or hovering over a UI element, like a context menu in picture 2.5.

In the manner of Dashboard, any abnormal state should be the first thing that attracts attention of a user.

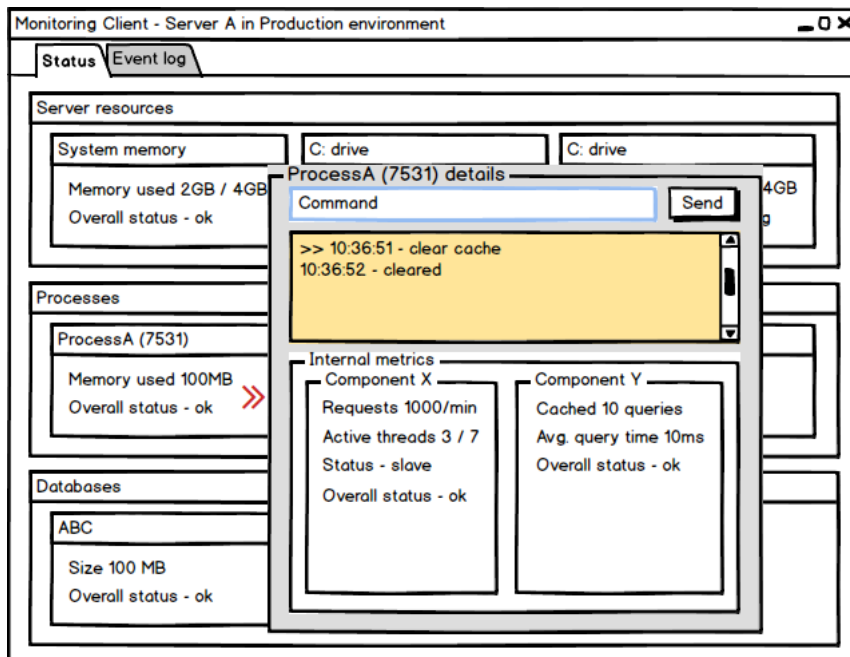


Figure 2.6: A process details view

2.3.3 Process details view

Sketch 2.6 shows an example of related information separation in order to prevent crammed views. In this case, information about a process are separated to "public" information showed in a server view and internals along with advanced functionality in a pop-up window.

2.4 Conclusion

I learned that it's important not to spend too much time in the design phase, but rather propose a viable solution and jump to implementation. There's always something that can be designed better but it can wait after the solution is out in open. Even if there are no new feature requests after the first iteration, users can change their mind about the current ones. The next chapter describes how the designed features do what they do and shows the sketches realized in colors.

Implementation

In comparison to design this chapter describes concrete technologies chosen to build a monitoring system as described and designed in previous chapters. It delves into details of important parts of the system, shows some code and final client's UI appearance. It starts with a section dedicated to third-party components used within the system and tools used during the development cycle. I use a title "Monitoring System" for the software output of this thesis for the rest of the document.

3.1 Technologies used

This section lists and describes software tools used in the development life cycle and third-party libraries used within Monitoring System.

3.1.1 Tools

Choice of tools used in development life cycle is as much important as the cycle itself because it can make different phases of the cycle more efficient and decrease amount of time consumed.

3.1.1.1 Visual Studio

<https://www.visualstudio.com/>

Visual Studio is almost a standard when developing .NET applications on Windows. There's no Integrated Development Environment (IDE) that could compete with what Visual Studio offers in terms of code writing and management. And there are many extensions available to make development a joy. Some I used are mentioned here.

3.1.1.2 JetBrains ReSharper

<https://www.jetbrains.com/resharper/>

3. IMPLEMENTATION

ReSharper is a Visual Studio extension that enhances development experience in areas such as code quality, error elimination, refactoring, code navigation and coding standards. There are some other competitive products like CodeRush <https://www.devexpress.com/products/coderush/> from DevExpress, but I'm used to ReSharper and other JetBrains tools that come packaged along with it.

3.1.1.3 JetBrains dotTrace

<https://www.jetbrains.com/profiler/>

dotTrace is a .NET performance profiler that runs as a standalone application but can also run as a Visual Studio extension so one can profile code without leaving IDE. It offers everything one expects from a performance profiler and does it in a nice clean user interface. It can even attach to a running process on a remote machine, which makes it ideal when dealing with unexpected process behavior.

I use it partly because it's a part of JetBrains ReSharper Ultimate package, but mostly because it's easy to use and understand. I describe some of its profiling options in section Performance testing.

3.1.1.4 JetBrains dotMemory

<https://www.jetbrains.com/dotmemory/>

dotMemory is a .NET memory profiler and what I write about dotTrace applies to dotMemory as well. It's an obvious choice if you already use dotTrace because their user interface is very similar. It surpasses other memory profilers thanks to automatic inspections which scans memory snapshots for most common types of memory issues and because of the ease of navigation through related objects in search for an issues origin. There's also dotMemory Unit which allows to extend .NET unit testing frameworks with the functionality of a memory profiler and check for memory issues while running unit tests.

3.1.1.5 Team Foundation Server

<https://www.visualstudio.com/products/tfs-overview-vs.aspx>

Team Foundation Server is a code-sharing, work-tracking and software-shipping product from Microsoft. It can be used for any language and integrates very well with Visual Studio. I used it for its VCS capabilities. It supports two different source control types – Team Foundation Version Control and Git, with the former being a centralized VCS and the latter being a distributed VCS.

3.1.1.6 Balsamiq Mockups

<https://balsamiq.com/products/mockups/>

Balsamiq Mockups is a small graphical tool to sketch out user interfaces for various types of applications. It's great for rapid prototyping thanks to focusing on structure rather than colors and basic interactivity options. Output can be in form of wireframes or sketches, which I find very pleasant to look upon because they really are as if drawn on paper. That's why I've chosen this tool.

3.1.1.7 Enterprise Architect

<http://www.sparxsystems.com.au/products/ea/>

Enterprise Architect from Sparx Systems is a modeling tool for various stages of development life cycle. It supports a wide range of modeling standards (like UML and BPMN) and provides powerful functions such as simulations, document generation, source code reverse engineering, project management and much more. It's not a lightweight tool by any means, so I used it only because it was available and I have some experience with it. It's targeted to more enterprise environment than brief UML sketching.

3.1.2 Frameworks and libraries

Some were an obvious choice, some I've chosen based on experience and the rest simply worked or not.

3.1.2.1 .NET Framework

<https://www.microsoft.com/net/default.aspx>

.NET is a general purpose development platform. It can be used for any kind of app type or workload where general purpose solutions are used. It has several key features that are attractive to many developers, including automatic memory management and modern programming languages. [12]

I've chosen it because I have a lot of experience with it, it works very well on Windows, it's already installed on all the machines where CAD is running, and portability to other systems is not required. Adjustment of the server parts to run on .NET Core is possible, should portability to other systems ever be required.

3.1.2.2 Windows Presentation Foundation

There are really only three sensible choices when developing a business desktop application targeting .NET Framework. They are listed in the order in which they emerged. Windows Presentation Foundation (WPF) was an obvious choice.

Windows Forms

[https://msdn.microsoft.com/library/aa983655\(v=vs.71\).aspx](https://msdn.microsoft.com/library/aa983655(v=vs.71).aspx)

3. IMPLEMENTATION

A bit old-fashioned but still used framework for UI development. It's considered finished in terms of adding new features, but there's a plenty of UI control packs available. It's based on even-driven architecture and Visual Studio provides clean What You See Is What You Get (WYSIWYG) editor with a wide range of built-in controls. Some of its drawbacks are limited styling, hard customization and very limited support for advanced graphics.

Windows Presentation Foundation

<https://msdn.microsoft.com/library/ms754130%28v=vs.110%29.aspx>

Instead of dragging controls into a window frame like in Windows Forms, WPF developers mostly rely on eXtensible Application Markup Language (XAML) to build UIs. Although WPF also supports event-driven development it's more often approached with MVVM pattern. It's strong where Windows Forms are not, but has its disadvantages. There are only few built-in controls (which stopped being a problem a long time ago thanks to many custom controls available on the internet), has steeper learning curve and won't run on Windows 2000 or lower.

Universal Windows Platform

<https://msdn.microsoft.com/windows/uwp/layout/design-and-ui-intro>

The latest trend in UI development on .NET platform with aim to work and look nice on any device running Windows 10 with minimal effort. It's also based on XAML but there are some differences and improvements to accommodate screens of any size and resolution, and deliver seamless touch experience.

3.1.2.3 protobuf-net

<https://developers.google.com/protocol-buffers/>

<https://github.com/mgravell/protobuf-net>

protobuf-net is a contract based serializer for .NET, that writes data in the "protocol buffers" serialization format. Protocol buffers are Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data — like XML, but smaller, faster, and simpler.

With metrics (and other things too) being sent over a network, some form of serialization is required. Protocol buffers are one of the fastest and most compact available, which is handy when communicating over a slow VPN.

3.1.2.4 clrzmq

<https://github.com/zeromq/clrzmq>

This library is basically a .NET wrapper around a native library called ZeroMQ or 0MQ <http://zeromq.org/>, which is a multi-platform library for socket communication. It offers communication patters like push-pull or publisher-subscriber and can be used from many languages thanks to a large community.

It was my first choice when searching for a lightweight socket server/client implementation. A server handling more clients at the same time and a client able to communicate with more servers at the same time, just what I needed. An extra native library was not an issue, but I didn't manage to get Monitoring service working when compiled in the release mode. Which was bad enough even before I discovered that I had to pack two Windows libraries with it in order to make it work on older Windows Server systems. That's when I started looking for another solution.

3.1.2.5 Helios

<http://helios-io.github.io/>

Helios is a library that makes it easy for .NET developers to create high throughput applications on top of network protocols like TCP, UDP, and more. It takes the complexity out of socket programming with intelligent Input/Output, concurrency, buffer management, and pipelining APIs.

With its event-driven architecture, Helios is really intuitive and easy to use, which was a pleasant change after struggling with clrzmq.

3.1.2.6 log4net

<https://logging.apache.org/log4net/>

log4net is a tool to help the programmer output log statements to a variety of output targets. In case of problems with an application, it is helpful to enable logging so that the problem can be located. With log4net it is possible to enable logging at run-time without modifying the application binary. The log4net package is designed so that log statements can remain in shipped code without incurring a high performance cost. It follows that the speed of logging (or rather not logging) is crucial. [13]

When comparing by the number of downloads from NuGet, log4net remains the most used logging framework for .NET applications. I've chosen it mainly because of that – more downloads usually indicate larger community, more tutorials and more questions answered in technical forums.

3.1.2.7 NUnit

<http://www.nunit.org/>

NUnit is one of many unit-testing frameworks available for .NET developers. ReSharper's support of this framework makes it an easy choice, although the latest version of ReSharper now boasts with out-of-the-box xUnit support, which is also a great unit-testing framework.

3.1.2.8 WiX Toolset

<http://wixtoolset.org/>

3. IMPLEMENTATION

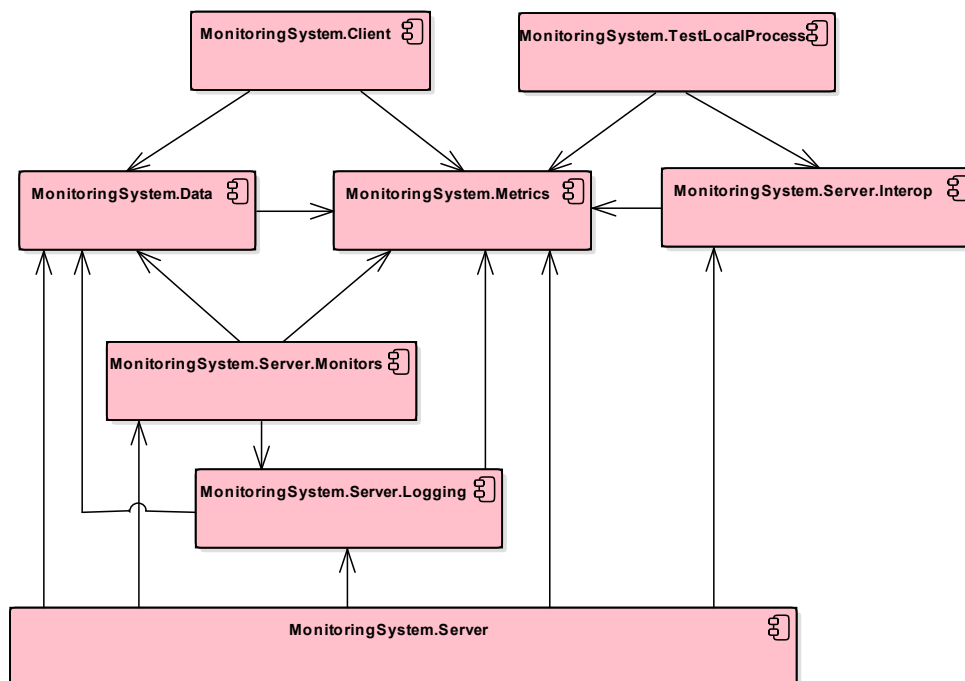


Figure 3.1: All the modules forming Monitoring System

The WiX toolset lets developers create installers for Windows Installer, the Windows installation engine. For purpose of this thesis, copy-pasting binaries would be enough to make the system work, but for deployments into various environments an installer configurable by a build server is a necessity. WiX lets one set up everything from an install location to permissions required for running, offers seamless product upgrade and downgrade, and it has an extension for Visual Studio.

3.2 Architecture

Diagram 3.1 shows all the modules in Monitoring System along with their dependencies. External dependencies are omitted for clarity. Each module represents a project in the solution and each project's output is either a class library or an executable file. Modules dependencies are explained in the following sections.

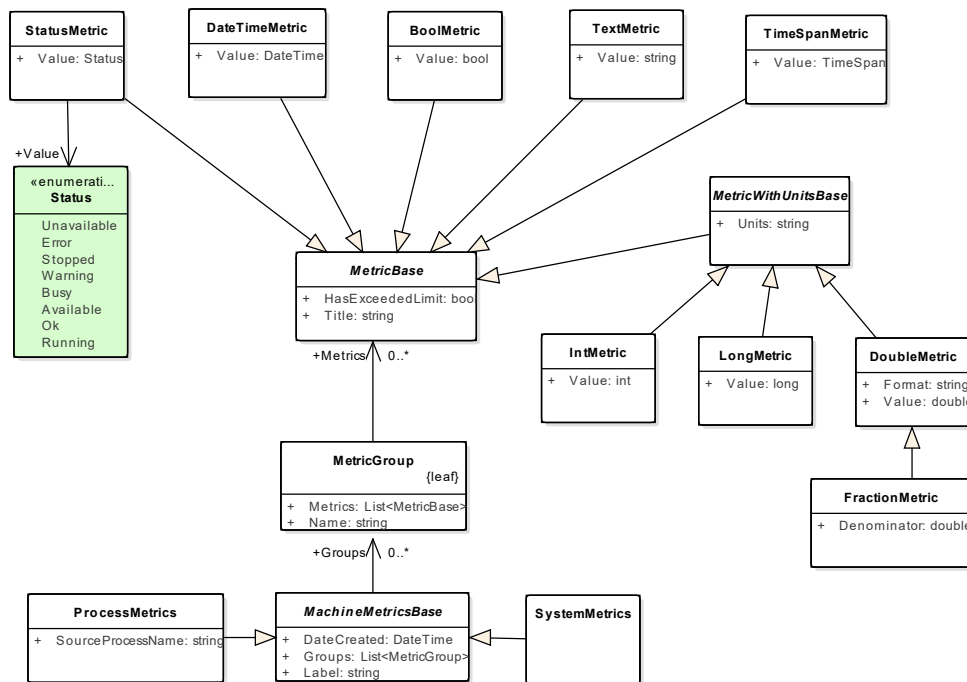


Figure 3.2: Metric types hierarchy

3.3 Metric system

Diagram 3.2 shows all the different metric types along with their dependencies. All these types are located in project `MonitoringSystem.Metrics`. The central piece is an abstract class called `MetricBase` which is a base class for all type specific metrics and defines a metric title and an indicator whether a value that a particular metric instance represents is out of an expected range. This range is not part of a metric; the property is set based on predefined rules by a monitoring service or by a service author.

All numeric metrics derive from an extra base class called `MetricWithUnitsBase` that enriches them with a property that allows to specify units in which a metric value is measured. The enumeration `Status` lists the most common expressions used when working with system resources.

3.4 Server

As the server part of Monitoring System logically consists of two parts I regard them as two sub-sections in this section.

3.4.1 Reporting module

The reporting module is a class library named "MonitoringSystem.Server.Interop" to signify that a monitoring service can interoperate with any .NET process without any further adjustments and configuration on the host process part.

It depends only on the metrics library, which I decided in order to make both sides deal with fixed data structures and therefore rely on business model restraints and static code analysis, to prevent mistakes in metric structure definition and parsing.

This library contains everything needed for communication using named pipes between a monitoring service and a process. The most important classes are `PollMetricServer` and `PushMetricClient`, their usage is in detail explained in section Reporting module. There's also `ManagementInterface` class, which I describe later in section Management Interface.

3.4.1.1 Polling model

When a service author decides to report metrics on request bases a monitoring service has to be given a name of processes that spawn from this service to poll them. Each of these processes is running a server (a dedicated thread) accepting requests through a named pipe, but service authors do not have to know how to name the pipe, they just start a `PollMetricServer` instance. The monitoring service then tries to connect to a named pipe for each running monitored process. Pipe names are composed from a hard-coded text and suffix that contains a process ID.

3.4.1.2 Push model

A monitoring service is running a named pipe server with predefined hard-coded name that service authors do not have to know. They just ask an instance of `PushMetricClient` to asynchronously push metrics to the underlying pipe.

3.4.2 Monitoring service

Most of the logic for Monitoring service can be found in project `MonitoringSystem.Server`. When a monitoring service starts, workers are registered in `WorkerManager` class which handles proper starting and stopping of all the workers. Each worker is implemented as a thread (or a long-running task, which is basically the same with a few minor differences).

3.4.2.1 Socket server

When a client connects to a server, the connection is kept open until the client disconnects. This logic is handled by `Helios`. Each request is prefixed with a

byte header, based on which the server determines how to process it and how should a response look like.

In the first iteration, while still using `clrzmq`, I started with the publisher-subscriber model to broadcast the latest metrics. Later on, I changed it to request-response model because all other communication with clients was done that way anyway. But the primary reason to change it were long-running tasks. This change made it easier to get results back to a client. The system of long-running tasks is more described in section Long-running tasks.

3.4.2.2 Scheduler

Class `WorkScheduler` maintains a set of tasks sorted by the date and time of their planned execution, from the earliest to the latest. This set is what I call an execution plan. Apart from public methods for tasks registration, the main scheduler logic is implemented in method `RepeatingAction`. The following code snippet is somewhat simplified version of the method.

```
void RepeatingAction ()
{
    SimpleTask dueTask = tasks.FirstOrDefault ();
    if (dueTask == null)
    {
        HasWorkToDo = false;
        return;
    }

    TimeSpan diff = dueTask.NextRunTime - DateTime.UtcNow;
    if (diff > TimeSpan.FromMilliseconds(150)) // 150ms tolerance
    {
        Thread.Sleep(500);
        return;
    }

    tasks.Remove(dueTask);
    Task.Factory.StartNew(dueTask.Execute)
        .ContinueWith(RescheduleRecurringTask, dueTask)
        .ContinueWith(ScheduleAnotherAttempt, dueTask,
            TaskContinuationOptions.OnlyOnFaulted);
}
```

In short, the scheduler inspects the earliest scheduled task and if it's the time (with a slight tolerance), the task is sent to be executed on a thread pool. If it's too early, the underlying thread is suspended for a moment and then it's all repeated. This ensures that the scheduler always picks the earliest.

3.4.2.3 Monitors, situations and reactions

All the monitor logic can be found in project `MonitoringSystem.Server.Monitors`. In the first iteration, each monitor was running on a dedicated thread and when started by Monitoring service, it started querying assigned monitor units

3. IMPLEMENTATION

and report metrics. However, since some units are not required to be checked more often than once in a minute, having so many threads, mostly in the suspended state, is still resource wasting. In the second iteration, they have been redesigned in a way that allows a data source check to be represented by a recurring task in a work scheduler.

Each monitor has its own configuration section in the service XML configuration file. When a monitor is not required, its section can be omitted. The following code snippet is a monitor section configuration example.

```
<processMonitor defaultCheckInterval="00:00:05">
  <processes>
    <passive name="devenv">
      <highCPU value="2.0" />
      <highMem value="1100 MB" />
    </passive>
    <passive name="totalcmd" checkInterval="00:01:00" />
    <passive name="MonitoringSystem.Server" />
    <active name="MonitoringSystem.TestLocalProcess">
      <exit actionId="emailToOndrej" />
    </active>
  </processes>
</processMonitor>
```

A monitor can specify a default check interval for all its monitor units. If it does not, the default is 10 seconds for all except a database monitor which defaults to one hour. Each unit can still override this setting by specifying its own check interval, like "totalcmd" process does in the example. A check interval defines how often is a resource going to be queried for metrics, both external and internal.

Processes can be configured either passive or active. This helps Monitoring service to decide whether to poll the process for metrics. As you can see, no other extra setting is required to collect metrics from a process. Setting a process incorrectly to active incurs only a tiny overhead which quickly diminishes over time until there's none.

By default, no resource specifies any situation. In example above, only two processes specify situations and only one of them is configured with a reaction. When there's no actionId attribute included in a situation, users of Monitoring client are still going to see what's wrong, but no automatic reaction takes place.

Reactions (or just actions in the configuration context) are configured in a different section of the configuration file. The one used referred in the example above can look like as following.

```
<email id="emailToOndrej"
  to="janacek.o@gmail.com"
  subject="Monitoring_Service_{0}" />
```

Subject and message attributes are optional as there are default values for them. When specified, they can leverage a few predefined parameter place-

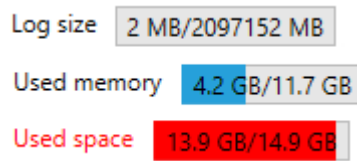


Figure 3.3: Fraction metric appearance examples

holders like `{0}` for a machine name. Different reactions have different attributes naturally. A situation can be matched only with a single action ID, but it can be a composite reaction which is composed from any number of specific reactions.

3.5 Client

Monitoring client is located in project `MonitoringSystem.Client` and depends only on the metrics library in order to understand received metrics and on `MonitoringSystem.Data` library which contains types and data used in communication between a client and a server. The project is structured in the MVVM fashion — models are located in the project folder `Models`, view models in `ViewModels` and views in `Views`. Folder `Views` contains only window views, however. Parts of the views are treated as user controls and are located in folder `Controls`.

New data from servers are pulled in a configurable interval asynchronously. When it's time, the client asks all connected servers for the latest metrics and they are processed as they arrive without blocking the UI thread (a thread responsible for a UI application responsiveness). They are stored in models which notify view models and view models in turn notify views using data binding.

A server node on Dashboard shows the most important information and its header is colored based on its overall status. This feature is explained in depth in section `Visibility of system status` in chapter `Testing`. For now, the most important information is the server status, when the last error occurred and what's the server time, all visible in screenshot 3.4.

WPF offers a powerful feature for displaying data called data templating. A data template defines how a data type should look like when asked to be displayed on a screen. So every metric data type, metric container data type and every view model has a data template assigned. Most of these templates are located in the root `App.xaml` file. The following code snippet shows example of how a data template for fraction metrics can look like and picture 3.3 shows possible outcomes.

3. IMPLEMENTATION

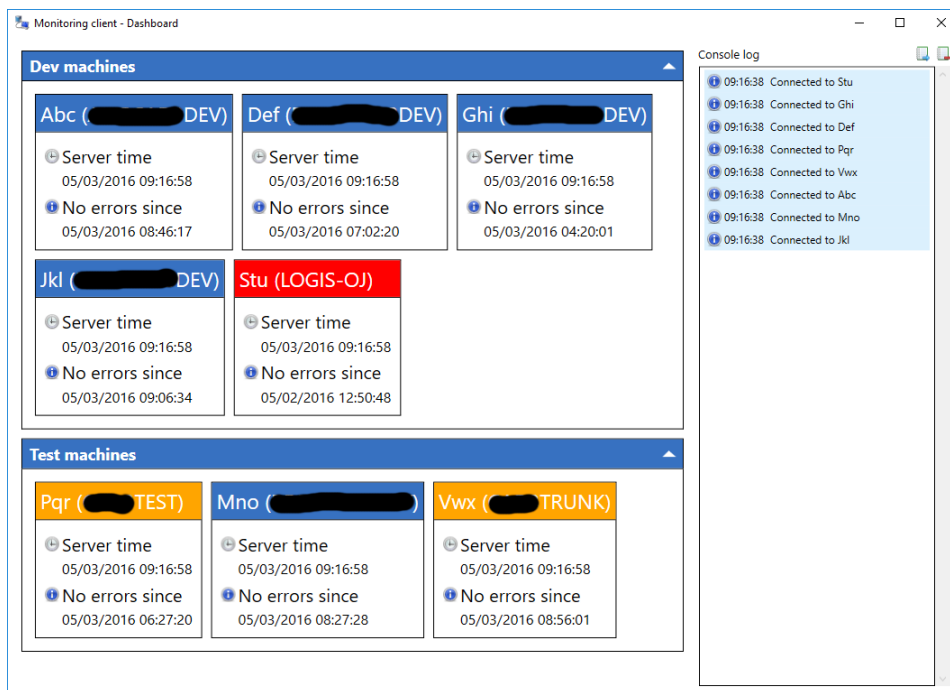


Figure 3.4: Dashboard implemented. Real machine names are concealed because of privacy policy.

```
<DataTemplate DataType="{x:Type metrics:FractionMetric}">
  <Grid>
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="Auto" />
      <ColumnDefinition Width="10" />
      <ColumnDefinition />
    </Grid.ColumnDefinitions>

    <TextBlock Text="{Binding Title}" />
    <Grid Grid.Column="2">
      <ProgressBar Maximum="{Binding Denominator}"
        Value="{Binding Value, Mode=OneWay}"
        Foreground="#26A0DA"
        BorderBrush="#BCBCBC" />
      <TextBlock Text="{Binding}"
        TextAlignment="Center"
        Padding="6,2" />
    </Grid>
  </Grid>
</DataTemplate>
```

There's a bullet point mentioning an existing platform for data visualization and querying in the beginning of section Existing solutions. That's the reason for not implementing any such logic into the client. Seeing the latest

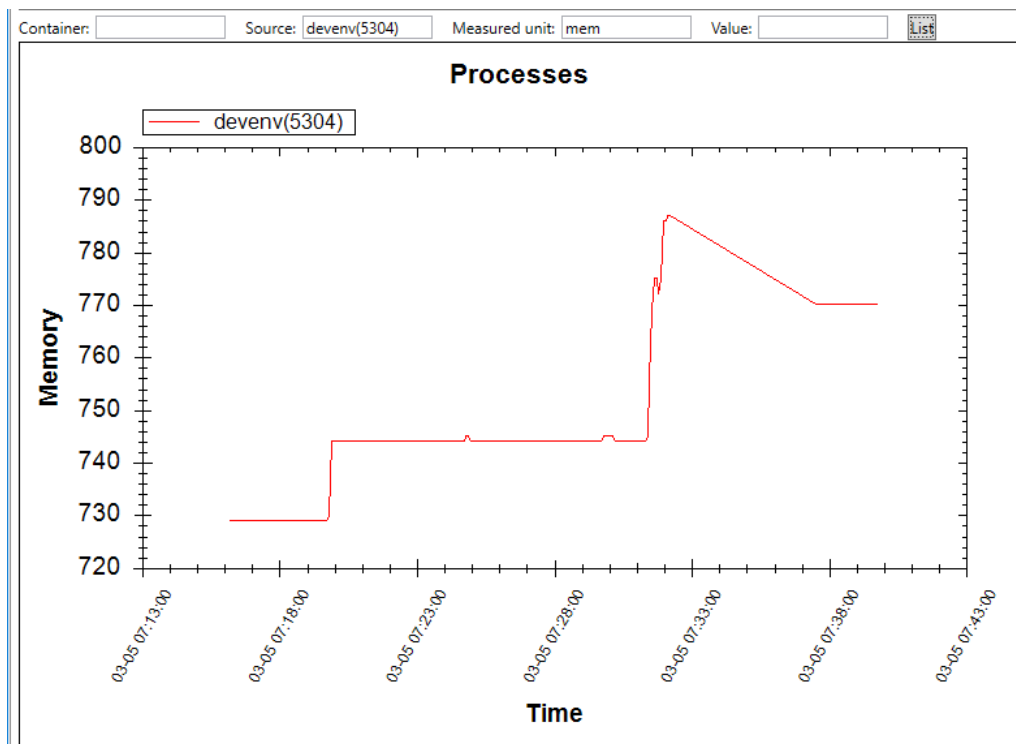


Figure 3.5: Metric history in Log Viewer

metrics is fine, but being able to view a history values for a particular metric is undoubtedly better. The production version of Monitoring client offers an extra behavior which allows to select a metric by clicking on it, leading to opening of an internal tool called Log Viewer and displaying the metric history as shown in picture 3.5.

More details about Monitoring client implementation decisions can be found in section Monitoring client UI.

3.6 Points of interest

This sections lists some of Monitoring System parts which I think are worth pointing out because of the value they brought to the whole system.

3.6.1 Console / Service host

Project MonitoringSystem.Server needs to run as a Windows service and there's a project type for Windows services in Visual Studio. However, if it was created as a Windows service project, testing would be hard and debugging nearly impossible, because a service has to be installed and registered

3. IMPLEMENTATION

in the computer first and does not have any standard output. Testing (in terms run it and see what it does) would be a tiresome process of following these steps:

1. Install the service only the first time.
2. Stop the service.
3. Copy over changed files.
4. Start the service.
5. Read log files to see what's happening with the service.
6. To debug, attach debugger to the service process and detach when finished.

This makes for annoying testing experience. One solution would be to move all the non-service related logic into a separate project and have two host projects — a Windows service project and a console application project. What I did instead is a single console project with all the service related code included and when the output executable is started, it checks whether it was started in user interactive mode and decides whether to run a console window or run as a service.

```
var controller = new ServerController();
if (Environment.UserInteractive)
{
    if (controller.Start())
    {
        while (Console.ReadLine() != "stop") { }
    }
    controller.Stop();
}
else
{
    ServiceBase.Run(new ServiceBase []
    {
        new WindowsService(controller)
    });
}
```

3.6.2 Management Interface

Management Interface is basically a named pipe server running on a dedicated thread inside a process. The pipe has a well-known name and is locally accessible to everyone. That does not pose a security issue because machines are very well secured and accessed only from a local network or over a VPN.

It allows run-time configuration of the hosting process, which can be very useful for testing of new features, performance tuning and maintenance, all

without need to restart the process. A service author decides what commands should be available via the interface, but there's a set of shared commands — list available commands, write a message to an event log and run Garbage Collector.

3.6.3 Long-running tasks

There are some potentially long-running tasks that a user can ask a monitoring service to carry out, create a memory dump, for instance. User initiates a long-running task and should be notified when it's done, either by a logged message or a change in the client UI state.

When a user requests such a task, a record is created in `LongRunningTaskRegister` class. It consists of

- a command text that represent the task,
- a name of the server carrying out the command and
- a callback (action to be carried out on the client when the task is finished).

After that a request is sent to the server where it's delegated to a work scheduler (described in section Scheduler) and marked with a flag symbolizing an external originator. Work like this, when finished, is reported back and when a status request comes from the originator, the results of the task is sent back. The client notices a received task result and asks the register to invoke a callback registered with the task. As of writing this document, registered tasks are not persisted.

3.7 Conclusion

I learned that choosing a more popular library over a less popular might not always be the best choice when looking for a component in an area one has not searched before. While it usually provides benefits of a larger community and more tutorials, it's not guaranteed to work best in one's particular scenario. To best avoid difficult situations when switching to a different library, encapsulating its logic is the best way to go.

Furthermore, it not always necessary to search for the most performing solution, premature optimization only delay a working product. It's better to choose some that simply works and discover if it's efficient enough by testing, which is what the next chapter is about.

Testing

Testing is an important part of every software development life cycle. Writing unit tests helps to prevent bugs in production by ensuring that tested units of code really do what they are expected to do. Integration tests take over where unit tests end by combining already tested units into components and testing these components, which reveals problems that occur only when isolated code units are combined. Then there are performance tests running on the whole system composed from all the components and testing the overall stability. And finally UI heuristics and testing ensuring that users won't get lost and are comfortable using the client application.

4.1 Unit tests

Unit testing is about testing individual functions that have no or very few external dependencies. They should be very fast to finish and produce results. I use them to test self-contained, error-prone units of code. There are a few different software development processes that rely on testing, such as test-driven development or behavior-driven development, but I don't tend to incline to any particular process. I use what feels right based on my experience.

All unit tests for Monitoring System are located in a project named "MonitoringSystem.Tests". Some I wrote before implementing the tested logic, some after, and I run them manually only when making changes in code they test.

A testing class has to be marked with an attribute [TestFixture] and a testing method with an attribute [Test] — these attributes apply for NUnit, which I use. Since NUnit support is not built into Visual Studio, at least a Visual Studio extension called "NUnit adapter" it required in order to see tests in the Test Explorer window. This extension is not needed for ReSharper users because ReSharper supports NUnit and tests show up in its Unit Test Sessions window. Nevertheless, having Visual Studio recognize unit tests brings a nice benefit — tests for a tested function show up right above the function in Visual

4. TESTING

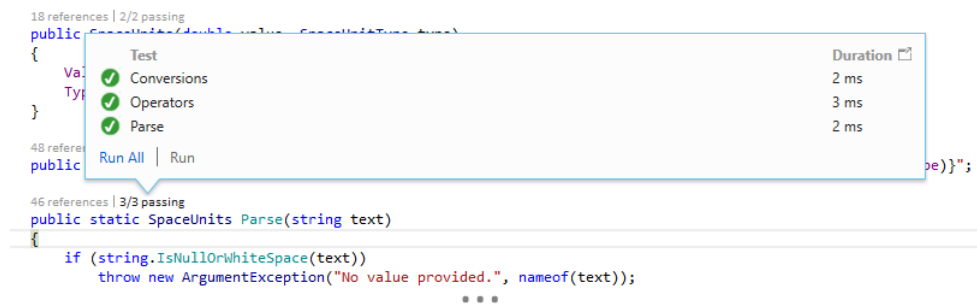


Figure 4.1: Using Code Lens with unit tests

Studio's feature Code Lens (showed in picture 4.1) and can be run from there as well.

There's a lot of code that works with system resources or values that change over time. These parts of code can't be very well tested with unit tests so that's where integration tests take over.

4.2 Integration tests

Integration between all components is tested by starting Monitoring service, Monitoring client and one or more processes reporting metrics. For testing purposes, I created a project named "MonitoringSystem.TestLocalProcess". It's a Windows Forms application with a few controls on it that allow change value of reported metrics, as shown in picture 4.2 (note that some values do not match because they change faster than the client updates). Just changing values and watching reactions of Management client indicates whether all these components work correctly.

4.3 Performance testing

JetBrains dotTrace supports 4 different profiling types: sampling, tracing, line-by-line and timeline. Most of the time, I use timeline because it shows call times same as the sampling but provides nice filtering based on an application run-time timeline (shown in picture 4.3). It can be filtered by threads, memory allocations and more.

When testing GUI applications, the most important information for me is how much time takes code running on the UI thread and what code runs there. For Monitoring service, utilization of created threads for workers is a nice metric and can be easily spotted under the timeline. When I saw it the first time when each monitor was running on its own thread, I realized that

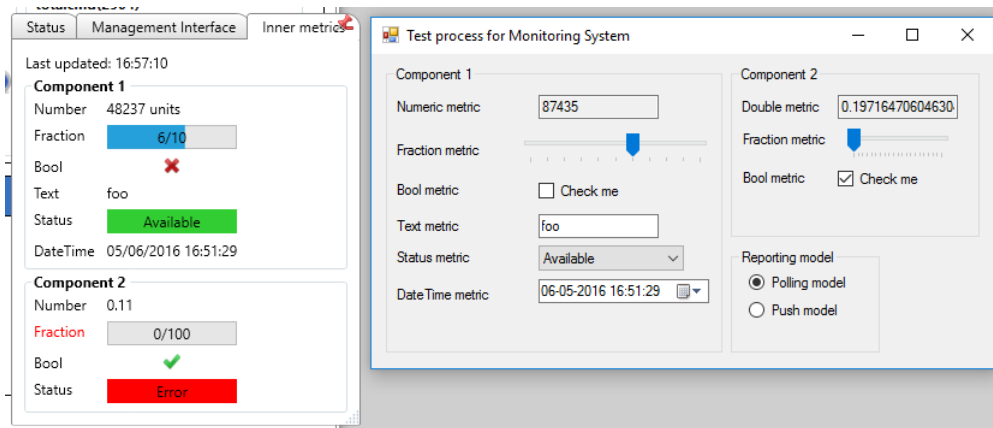


Figure 4.2: Integration testing using a test process

it could really all run on the same thread. This idea led to creating a work scheduler.

When there's a part of code that has high call time, I usually use tracing method, which shows an exact number of calls. If the number looks like it should be smaller, there might actually be a problem caused by calling some method too often. In a situation like this, caching results is usually a good way to go, unless the method was accidentally called more than once in a single operation.

JetBrains dotMemory provides only timeline memory profiling, but all that's important is in there. In comparison to dotTrace timeline profiling, this one is real-time. It's possible to run an application with attached profiler on the side and watch the timeline as you initiate different functions of the application. A common scenario for GUI applications is opening and closing the same window over and over again, to see if there are any resources that are not freed after the window is closed, leading to memory leaks. A timeline example is shown in picture 4.4.

As I mentioned earlier in this document, dotMemory shines when it comes to auto-inspections. When a memory snapshot is created, it shows what types are associated with the largest used memory blocks, how much memory is wasted by string duplicates, big arrays that are nearly empty and more. For example, if a lot of event log entries was fetched from a monitoring service to a client and most of them had the same message, there could potentially be a large amount of the same strings. Usually, taking a snapshot is enough to see if drilling down is necessary.

4. TESTING

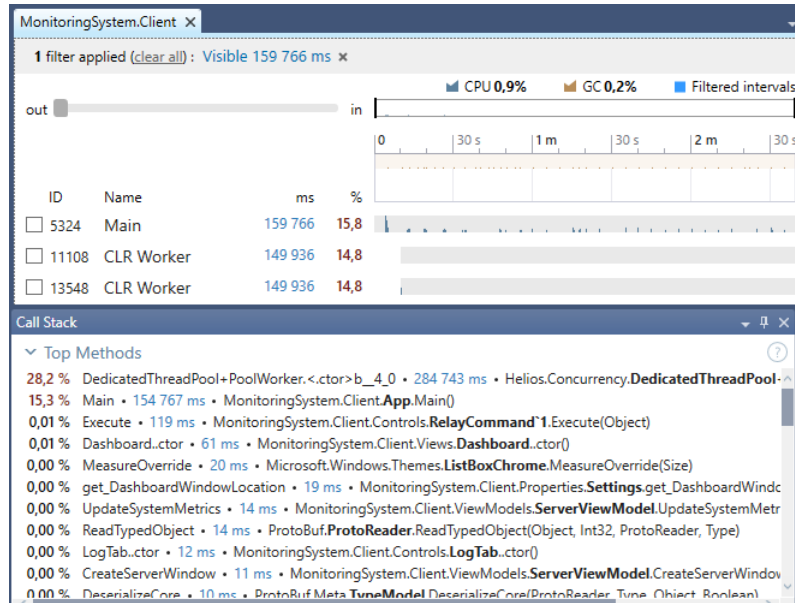


Figure 4.3: Monitoring client performance tracing with timeline

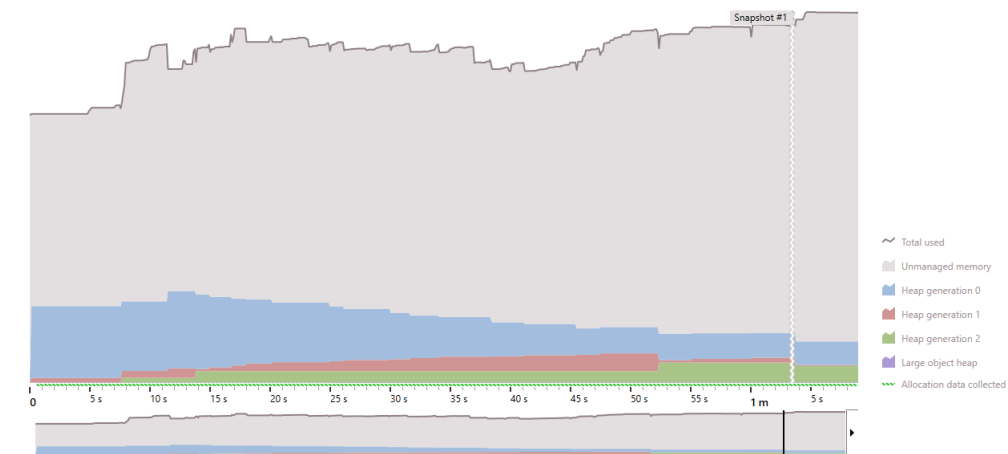


Figure 4.4: Monitoring client memory profiling on a timeline

4.4 Monitoring client UI

This section is dedicated to user interface testing. I have not created any automated UI tests, however. The GUI appearance follows sketches from chapter Design and was tested and improved by using Nielsen's usability heuristics and after that presented to users for user testing on real scenarios listed in section Use case scenarios.

This section is composed from subsections which represent all Nielsen's principles for interaction design. Each subsection starts with a principle description paragraph (all copied from <https://www.nngroup.com/articles/ten-usability-heuristics/>) followed by images and/or description of how I addressed it.

4.4.1 Visibility of system status

The system should always keep users informed about what is going on, through appropriate feedback within reasonable time.

4.4.1.1 Problematic server

Whenever an issue occurs on a server, its node header on Dashboard changes color according to the issue severity. It's orange for a warning, red for an error and blue if everything is alright. Example can be seen in picture 3.4. The issue can be easily traced by following these steps:

1. Open the server details window.
2. Look for a metric category (System resources, Processes, etc.) with orange/red header.
3. Once inside the category, look for an item with orange/red Overall status or Status when the former mentioned is not present.
4. The problematic metrics' labels are colored red. In case of processes, if no label is red, open the process details and see Inner metrics tab. You are going to find the problematic metrics there.

Steps 2 and 3 are shown in screenshot 4.5. Step 4 is shown in picture 4.6.

4.4.1.2 Long-running tasks

When a user initiates a long-running task, a progress ring appears in an appropriate place and disappears when the task is finished. The result is visible in the place of the task origin (fetching event log entries, for instance) and/or in the event log panel on Dashboard and/or a taskbar pop-up appears. Image 4.7 shows indicator of an ongoing process memory dump operation and image 4.8 a taskbar pop-up with a result.

4. TESTING

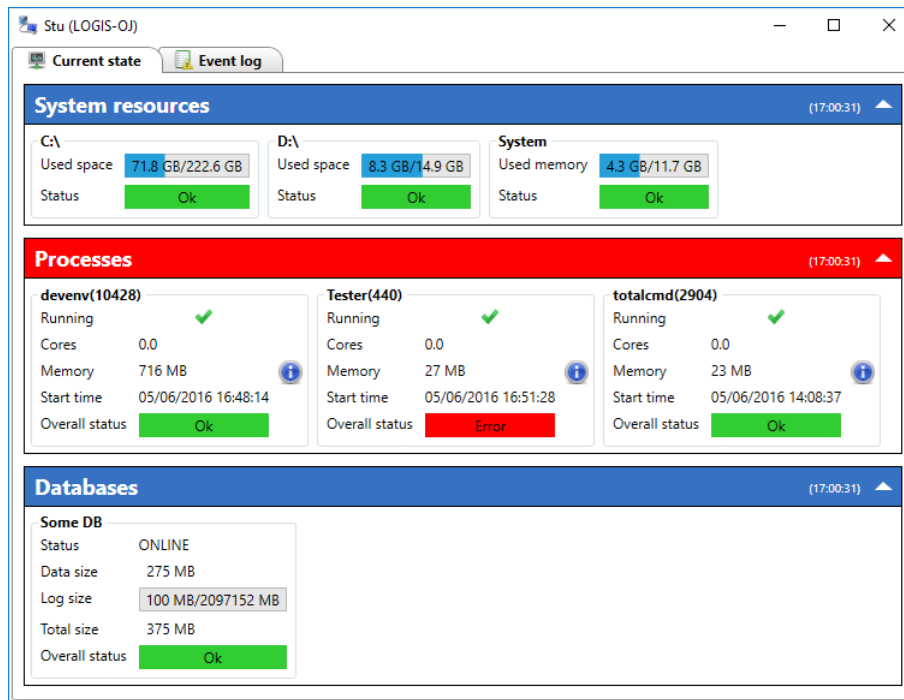


Figure 4.5: A resource causing an issue

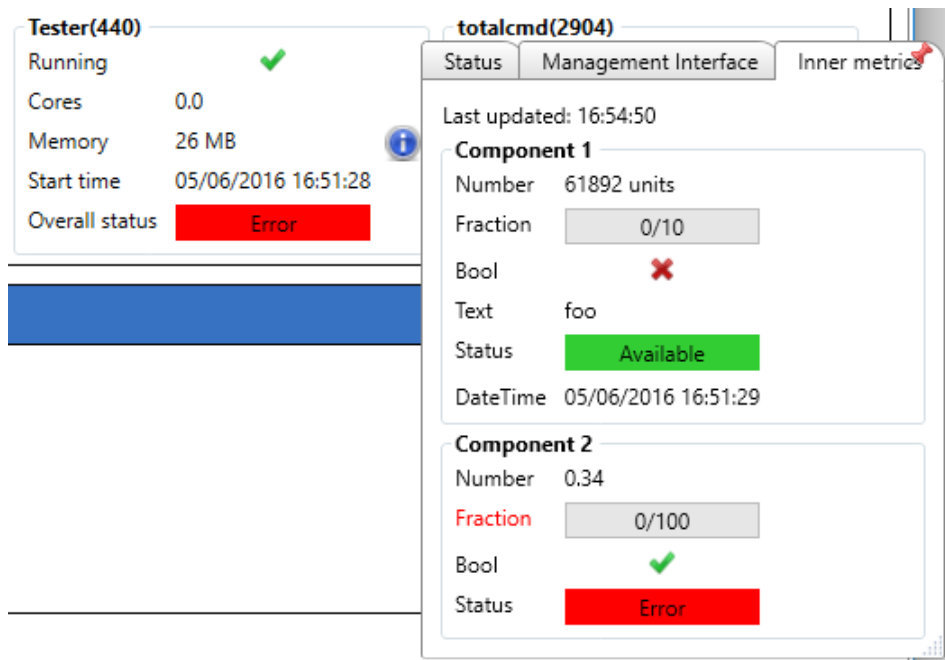


Figure 4.6: Metrics causing an issue

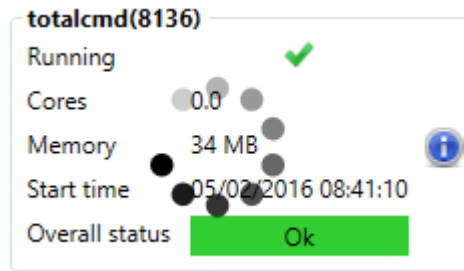


Figure 4.7: Ongoing memory dump

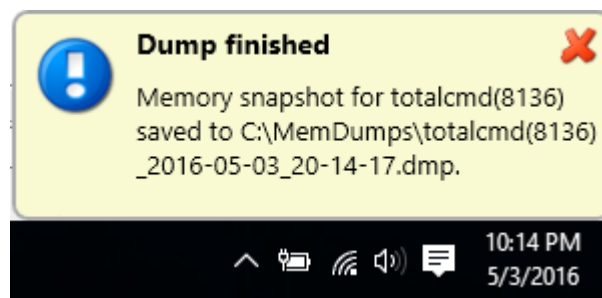


Figure 4.8: Memory dump finished

Admittedly, the pop-up could point to a taskbar icon of the application or at least mention "Monitoring client" in the header so that it's not confused with a message from a different application. It could also integrate with Action center in newer versions of Windows.

4.4.2 Match between system and real world

The system should speak the users' language, with words, phrases and concepts familiar to the user, rather than system-oriented terms. Follow real-world conventions, making information appear in a natural and logical order.

- All users are technically oriented and well acquainted with terms used in the client.
- Familiar icons are used to emphasize meaning of some labels or even fill in for them.
- Server nodes in environments are ordered alphabetically same as resources in resource groups in a server details view.
- Colors emphasize problems severity.

4.4.3 User control and freedom

Users often choose system functions by mistake and will need a clearly marked "emergency exit" to leave the unwanted state without having to go through an extended dialogue. Support undo and redo.

No operation is blocking. If one requests event log entries, but decides otherwise while they are being fetched from a server, one does not have to wait on the event log tab and can simply leave with a possibility to come back later and see the entries.

4.4.4 Consistency and standards

Users should not have to wonder whether different words, situations, or actions mean the same thing. Follow platform conventions.

- In some places an icon emphasizes meaning of the adjacent label, in other places like a process details pop-up, a pin icon does exactly what a user would expect it does.
- Whenever there's a progress ring (a progress bar in the circular shape) it always means an ongoing operation. See picture 4.7 for example.
- Some labels are actionable. When a mouse cursor passes over them, the mouse cursor changes from an arrow to a hand.

4.4.5 Error prevention

Even better than good error messages is a careful design which prevents a problem from occurring in the first place. Either eliminate error-prone conditions or check for them and present users with a confirmation option before they commit to the action.

When some functionality is not available on the background, the related parts of UI are disabled. This way it's not possible to filter event log entries when they are being updated, ask for a process memory dump when a server is not responsive or there's already an ongoing dumping operation on that process.

4.4.6 Recognition rather than recall

Minimize the user's memory load by making objects, actions, and options visible. The user should not have to remember information from one part of the dialogue to another. Instructions for use of the system should be visible or easily retrievable whenever appropriate.

There are only two windows in the whole Monitoring client and no dialogs. When a user opens details for a server node, the window with the details has a title specifying which server details it contains as shown in picture 4.9.

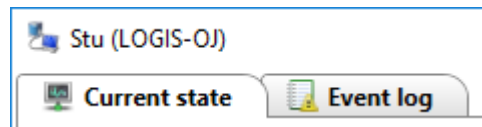


Figure 4.9: A server view window title indicates which server details are opened

4.4.7 Flexibility and efficiency of use

Accelerators – unseen by the novice user – may often speed up the interaction for the expert user such that the system can cater to both inexperienced and experienced users. Allow users to tailor frequent actions.

Event without demonstrating usage of the client users were able to perform basic tasks like checking a specific value on a specific server and fetching event log entries. They agreed that the user interface is very intuitive.

However, there are some advanced features that are not visible at first sight and can make difference for experienced users.

- A dashboard window and a server view remember their position on screen for future use.
- When a dashboard window is closed, every child window (server view windows) are closed along with it, so there's no need to close them one by one when finished working with the client.
- Advanced functions related to monitored resources are hidden in a context menu rather than being exposed via icons directly on resources.
- The last error from a server can be displayed by hovering over the information in the server node.
- Some process commands allow auto-complete so it's not necessary to write the whole command name.

4.4.8 Aesthetic and minimalist design

Dialogues should not contain information which is irrelevant or rarely needed. Every extra unit of information in a dialogue competes with the relevant units of information and diminishes their relative visibility.

The tile design is very simplistic. Information relating the same thing are strategically distributed among a few places that are logically linked together and allow fast and easy drilling down. Less used features always give place to more frequently used ones.

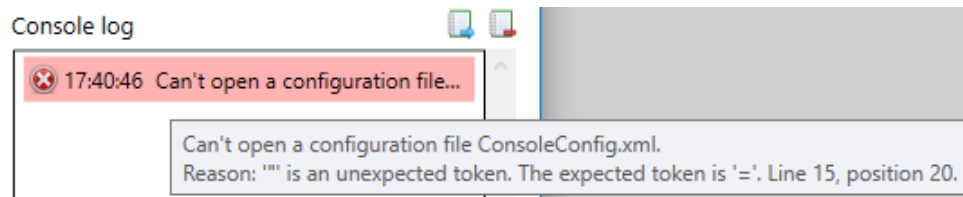


Figure 4.10: Client error with details

4.4.9 Help users recognize, diagnose, and recover from errors

Error messages should be expressed in plain language (no codes), precisely indicate the problem, and constructively suggest a solution.

As an example, when there's something wrong with a configuration file, the application starts up just fine and writes a message to the log panel as shown in picture 4.10.

4.4.10 Help and documentation

Even though it is better if the system can be used without documentation, it may be necessary to provide help and documentation. Any such information should be easy to search, focused on the user's task, list concrete steps to be carried out, and not be too large.

This is something that's presently missing, but is definitely going to be implemented in the next iteration.

4.5 Conclusion

Apart from UI usability tests which I did only once at the end, I closely watched performance from start to end and run tests after implementing any major feature. There might still be some hidden issues that will surface later, but running for a few months so far haven't brought any critical issues which I attribute to my testing strategy.

Conclusion

To evaluate whether my work meets the thesis target I review the assignment statement by statement.

The objective of the thesis is to design and implement a system, built on the .NET platform, that will monitor customer's servers state and mainly proprietary .NET processes running there, collect data (metrics), ... The output of this thesis is a working monitoring system built on .NET Framework. Monitors allow monitoring of different server resources representing the server state.

... allow to set up actions to be executed when certain conditions are met, ... This sophisticated system is described in implementation subsection Monitors, situations and reactions.

... and display the collected metrics to users. A Monitoring client screenshot 4.5 testifies in favour of that statement.

In cooperation with the supervisor, specify metrics to be monitored and collected by the server part and rules that allow to specify the server part reaction to certain metrics values. Both metrics and reactions are listed in subsection Server part of chapter Analysis. However, metrics are not specified directly (only by listing metrics sources) because reactions are described in a way that clearly implies what metrics need to be collected.

Test and document the whole system. I performed a variety of tests after finishing every major feature and written about it in chapter Testing. With documentation, I always strive for self-documenting code and where the intent is not clear enough a code comment is present. Options in configuration files for both Monitoring client and Monitoring service are described in sample configuration files included in their installers. Installation and user guides can be found among appendices.

Some sections indicate that the work was done in iterations and so it was. Objectives and requirements were pretty clear right from the start so analysis was repeated only once to ensure that are requirements were taken into consideration. The rest of the development life cycle phases were repeated

over and over again until all of them yielded desired results.

Moving forward

This work primarily helped me to reinforce my previous experience in software development – importance of having a clear target and requirements, releasing a viable product as soon as possible and not underestimating testing.

Logis support team appreciates not having to remote desktop to machines just to check event logs. They would like to have an option to filter event log messages based on severity. Developers appreciate metric reporting from their services because in combination with Log Viewer they can now compare performance history with events that were happening during that time. They also love Management Interface which is a new standard of run-time process configuration and maintenance. They would appreciate if finished process memory dumps could be automatically downloaded to their machine when finished.

What's next

There are many ideas as what to do next with Monitoring System, starting from minor features like a help button in Monitoring client (top right corner in picture 4.11) with tips like keyboard shortcuts. Adding a list of available client configurations (just above event log panel in picture 4.11) would make usage definitely more pleasant for users who often switch in between them. More automation like a memory dump reaction — when a hanging process is detected it would take a hang dump and restart without waiting for a user. It's a very rare situation, but should it happen, a swift resolution is important.

The following idea originated from features like sending commands to processes and fetching an event log — Monitoring System could not only serve for monitoring and reporting, but become a hub for remote product management where monitoring and reporting play key roles. Adding a layer for extensibility could allow create custom plugins, file transfer support could allow download created memory dumps and upload a new product version. This would be a first step towards remote deployments and hot patches.

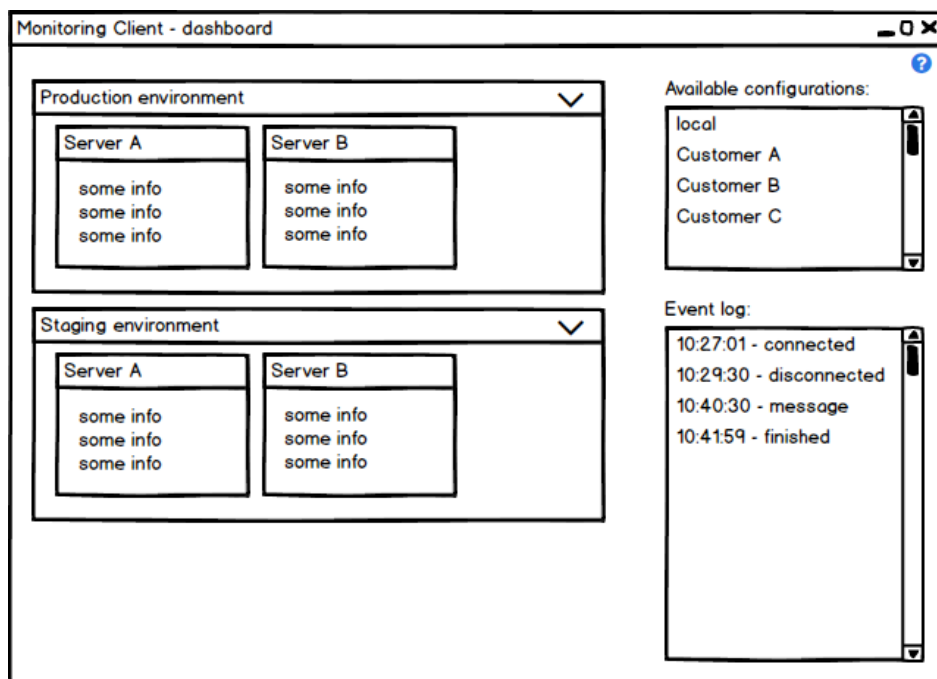


Figure 4.11: Dashboard design showing new features for future development

Bibliography

- [1] Wikipedia. Monitoring. [cit. 2016-04-17]. Available from: <https://en.wikipedia.org/wiki/Monitoring>
- [2] Microsoft. Introduction to Windows Service Applications. [cit. 2016-04-18]. Available from: <https://msdn.microsoft.com/library/d56de412%28v=vs.110%29.aspx>
- [3] Elastic. An Introduction to the ELK Stack. [cit. 2016-04-20]. Available from: <https://www.elastic.co/webinars/introduction-elk-stack>
- [4] Elastic. The Elastic Stack. [cit. 2016-04-20]. Available from: <https://www.elastic.co/products>
- [5] Mark Seger. Collectl. [cit. 2016-04-20]. Available from: <http://collectl.sourceforge.net/index.html>
- [6] Microsoft. Overview of Windows Performance Monitor. [cit. 2016-04-20]. Available from: <https://technet.microsoft.com/library/cc749154.aspx>
- [7] Microsoft. Windows Management Instrumentation. [cit. 2016-04-20]. Available from: <https://msdn.microsoft.com/library/windows/desktop/aa394582%28v=vs.85%29.aspx>
- [8] Microsoft. Monitoring Performance Thresholds. [cit. 2016-04-20]. Available from: [https://msdn.microsoft.com/library/wz3d1wc5\(v=vs.90\).aspx](https://msdn.microsoft.com/library/wz3d1wc5(v=vs.90).aspx)
- [9] Fowler, M. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, first edition, 2002, ISBN 0321127420.
- [10] Microsoft. Interprocess Communications - Pipes. [cit. 2016-04-22]. Available from: [https://msdn.microsoft.com/library/windows/desktop/aa365780\(v=vs.85\).aspx](https://msdn.microsoft.com/library/windows/desktop/aa365780(v=vs.85).aspx)

BIBLIOGRAPHY

- [11] Microsoft. Named Pipes. [cit. 2016-04-22]. Available from: [https://msdn.microsoft.com/library/windows/desktop/aa365590\(v=vs.85\).aspx](https://msdn.microsoft.com/library/windows/desktop/aa365590(v=vs.85).aspx)
- [12] Microsoft. .NET Development. [cit. 2016-04-24]. Available from: <https://msdn.microsoft.com/library/ff361664%28v=vs.110%29.aspx>
- [13] NuGet. log4net on NuGet. [cit. 2016-04-24]. Available from: <https://www.nuget.org/packages/log4net/2.0.5>

Acronyms

API Application Programming Interface.

CAD Computer Aided Dispatch.

GUI Graphical User Interface.

IDE Integrated Development Environment.

MVVM Model-View-ViewModel.

OS Operating System.

RAM Random Access Memory.

UI User Interface.

VCS Version Control System.

VPN Virtual Private Network.

WMI Windows Management Instrumentation.

WPF Windows Presentation Foundation.

WYSIWYG What You See Is What You Get.

XAML eXtensible Application Markup Language.

Installation guide

This appendix describes how to get Monitoring System up and running on your machine, for testing purposes, as comfortably as possible.

1. Make sure that Microsoft .NET Framework 4.5.1 or higher is installed (check in Programs and Features on your PC). If not, download and install from <http://smallestdotnet.com/>.
2. Open the contents of the enclosed DVD in a file explorer.
3. Navigate to folder install/installers and run MonitoringSystem.ClientSetup.msi and MonitoringSystem.ServerSetup.msi. This will install both a client and service to your machine. They will be located in the Program Files folder under OjMonitoringSystem. Do not start any of them yet.
4. Copy the whole folder install / testprocess to your PC and run the application inside by starting MonitoringSystem.TestLocalProcess.exe
5. Navigate to folder install / configuration and copy Service.xml to the folder Service where the service is installed. Do the same with ConsoleConfig.xml for the client.
6. You can start the service as a console application by running Service / MonitoringSystem.Server.exe from a file explorer, or start the service using Services app in Windows or tab Services in Task Manager. The service is named OjMonitoringService.
7. Start the client by running Client / MonitoringSystem.Client.exe.
8. Everything should be running and you should be able to see what picture B.1 shows. Now you can play with it.
9. To uninstall, go to Programs and Features, find OJ Monitoring Service/Client and uninstall.

B. INSTALLATION GUIDE

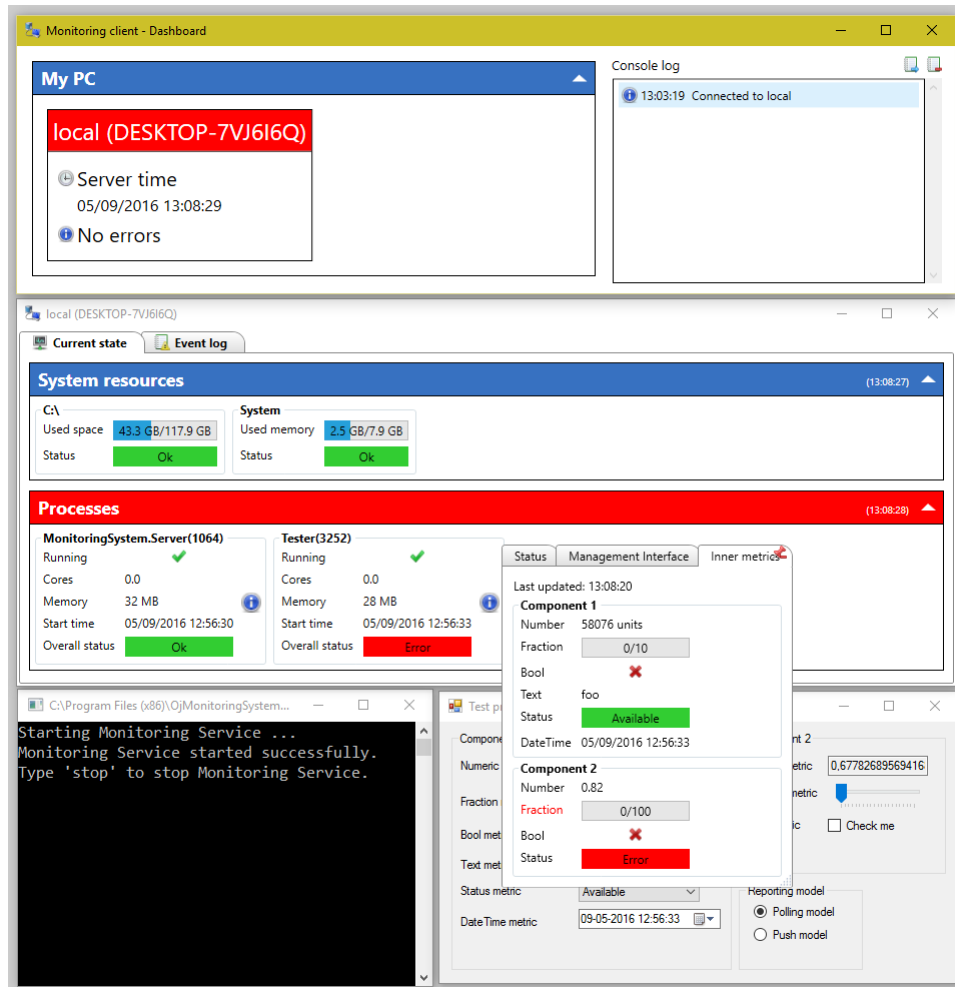


Figure B.1: When everything is running after installed

User guide

There are two guide sections. The first one is an usage/integration guide for developers who want to leverage capabilities of Monitoring System by sending metrics from their services. The second one is Monitoring client tutorial.

C.1 Reporting module

1. Add a reference to `MonitoringSystem.Server.Interop.dll` and `MonitoringSystem.Metrics.dll` from your project.
2. Choose a metric publishing model:
 - Push model — create an instance of `PushMetricClient` type. When you see it fit to send metrics to a monitoring service, create an instance of `ProcessMetrics` type, and call `TrySendAsync` method on the `PushMetricClient` instance providing the metrics. If the result is false, see `LastResponse` property on the `PushMetricClient` instance for errors.
 - Poll model — create an instance of `PollMetricServer` type providing a name that will represent your application in Management client processes, an instance of a log4net logger and finally a method delegate that generates metrics. Start the polling server by calling `Start` method on the `PollMetricServer` instance.

C.2 Monitoring client

This is a step by step description of Post-deployment support scenario which involves usage of nearly all of Monitoring client features.

1. To load an environment for a certain customer, either include the right configuration in `ConsoleConfig.xml` or run `MonitoringSystem.Client.exe` with an argument – a configuration file path.

2. Let's say there's a server with an orange or red header.
3. Click the server title, the server details window opens.
4. Let's say there's a process with an orange or red header.
5. In the current window, switch to Event log tab, type in the name of the event log where the process sends messages (real users know) and click Attach to log button. Using filters, find if there are any information that can help you diagnose the problem.
6. Return to Current state tab and do what's necessary. That might be:
 - Take the process memory dump. The feature is available after you right-click the process, a context menu opens and there's an option to dump memory.
 - Open the process details by clicking its information icon on the right and see Inner metrics tab, if the process reports any.
 - Switch to Management Interface tab to send commands that could help with the problem. If you don't know available commands, type "?" and send it.
7. When you are done and want to close everything, close Dashboard window to close all open windows at once.

Contents of enclosed DVD

	readme.txt.....	a brief description of the DVD contents
	code.....	source code of the Monitoring System
	design....	files produced by Balsamiq Mockups and Enterprise Architect
	install	
	configuration.....	prepared configuration files
	installers.....	Monitoring client and service MSI installers
	testprocess.....	an application used for integration testing
	text.....	the thesis text and source files
	DP_Janacek_Ondrej_2016.pdf.....	text of the thesis in PDF format
	source.....	LaTeX source files, images and contents of the thesis