Project Title

*Modifiability and Availability in maturing Software architecture*

Project period
February 2018 – June 2018 (part time)

Author
Anders Hvidgaard Poder, ahp@acct.dk
Student no.: 951439

Supervisor
Henrik Bærbak Christensen
Table of content

Project Title .............................................................................................................................................. 1

Project period ........................................................................................................................................... 1

Author ...................................................................................................................................................... 1

Supervisor .................................................................................................................................................. 1

1 Motivation .............................................................................................................................................. 4

2 Problem formulation ............................................................................................................................. 7

2.1 Scenario 1 – Client REST service becomes non-response............................................................... 7

2.2 Scenario 2 – Control Service needs to have extended functionality .............................................. 7

2.3 Hypothesis 1 ...................................................................................................................................... 9

2.4 Hypothesis 2 ...................................................................................................................................... 9

3 Method ................................................................................................................................................ 10

3.1 Hypothesis 1 ..................................................................................................................................... 11

3.2 Hypothesis 2 ..................................................................................................................................... 11

3.3 Deployment ...................................................................................................................................... 11

3.4 Definitions ....................................................................................................................................... 12

4 Tools .................................................................................................................................................... 12

4.1 Lock Simulator ................................................................................................................................ 12

4.2 Client REST test double .................................................................................................................. 14

4.3 Performance and Availability evaluation tool (PAAET) ................................................................ 15

5 Analyses and Results .......................................................................................................................... 16

5.1 Scenario 1 – Client REST service becomes non-response ............................................................... 16

5.1.1 Pattern design and implementation .......................................................................................... 17

5.1.2 Test data ..................................................................................................................................... 21

5.1.3 Test execution ............................................................................................................................. 22

5.1.4 Evaluation .................................................................................................................................. 25

5.2 Scenario 2 – Runtime service update ............................................................................................... 26

5.2.1 Pattern design and implementation .......................................................................................... 29

5.2.2 Test data ..................................................................................................................................... 33

5.2.3 Test execution ............................................................................................................................. 34

5.2.4 Evaluation .................................................................................................................................. 37

6 Conclusion .......................................................................................................................................... 38

7 References ............................................................................................................................................ 39

8 Appendix A – Realistic message activity ............................................................................................ 40

9 Appendix B – Communication choices ............................................................................................... 40

9.1 Internal communication channels .................................................................................................. 40
9.2 External communication channels ........................................................................................................... 40
10 Appendix C – Tool description and configuration ....................................................................................... 40
10.1 Lock simulator ........................................................................................................................................... 41
10.2 Performance and Availability evaluation tool (PAAET) ............................................................................ 41
11 Appendix D – Test and configuration data .................................................................................................... 42
12 Appendix E – Race condition test .................................................................................................................. 45
13 Appendix F – History of external functionality .............................................................................................. 48
14 Appendix G – IPC single image diagram ...................................................................................................... 49
15 Appendix H – UnitTest and results ................................................................................................................ 49
15.1 SlACircuitBreaker ..................................................................................................................................... 50
15.2 SBCConnectionRepository ....................................................................................................................... 51
15.3 DBCachedExecution ................................................................................................................................. 51
16 Appendix I – REST communication considerations ..................................................................................... 51
17 Appendix J – Selected graphs ....................................................................................................................... 52
17.1 Scenario 1 CPU with no delay ................................................................................................................... 53
17.2 Scenario 1 Memory as REST received delay increase, 15msg/s .............................................................. 53
1 Motivation

“It is difficult to predict, especially about the future” [1]. One of the biggest challenges facing any developer is knowing what quality attributes to focus on. What is due diligence, what is premature optimization and what is gold plating? One might know what is important now, but what will be important in the future? This is the very nature of “Design for redesign”.

Access Technology was created to supply online access control (OAC) for buildings or fenced enclosures via the GSM network.

A common user story is a company with a small number of gates and/or doors, possibly geographically separated. By using the GSM network for communication, the installation is much simpler, but still achieves the same functionality as a cabled system. After installation the customer uses the hosted Website control panel to create users and access codes as well as decide on the status of the door (opening times and which users and access codes may open it). The users can then call the Lock, send an SMS, type on the key pad, etc. to unlock it. The Lock sends all actions to the Control Service.

The initial deployment diagram can be seen in Figure 1:

![Initial architecture diagram]

The locks use TCP connections to communicate, and all connections are from the locks inwards. The SMS and E-Mail notification is used exclusively to notify the client (end customer or service technician) about issues needing attention. The locks maintain a continuous connection to allow for commands being sent unsolicited to the lock, and the locks contain a local copy of the relevant data relating to permissions and users, to allow for temporary loss of connectivity.

Events and notifications are transmitted to the server instantly or cached locally if connectivity is temporarily lost. Upon loosing and restoring the connection the Control Service exchanges state information and permissions with the lock to ensure that no information was lost during the lost connection.

The architecture was designed with these key Quality Attributes [2] in mind:

- **Availability**
  The Control Service and Database must be highly available, and tactics [2] used to achieve this is; Fault detection, Fault recovery (Passive redundancy) and Fault Prevention.
The Availability requirements for the Website control panel is much less strict and no special Availability efforts have been implemented.

The Control Service writes relevant messages to the database and is responsible for maintaining the part of the database containing the access codes and status of the locks. This is chosen to ensure that the Website control panel does not hold locks to Control Service tables. The Website communicates desired changes to the service which process and persist them. The Control Service receives these communications in a single thread with limited priority, to prevent the Website control panel from lowering the Control Service availability in any significant way.

- **Performance**
  The Control Service must be able to support many (thousands) of simultaneous connections and even more messages, either internal messages in the service resulting in communication with the Locks, or messages from the locks. Due to the use of the 2G GSM network the communication with the locks must be minimized as it is quite expensive (monetary) and slow (many pending delivery confirmations). This is achieved by using a proprietary minimal protocol, employing piggybacking (grouping messages into a single TCP message to limit the number of TCP headers and TCP ACK) and minimizing reconnects (reconnects require several messages and extra overhead due to GSM setup and tear-down).

  The Control Service furthermore maintains a cache of the access codes and states in RAM to allow fast responses to messages or connections from the locks. The Control Service is highly multithreaded (high concurrency - Resource management [2]). The Control Service is executed in a single process, which allows for simple Shared Memory Communication. Simple Shard Memory Communication is defined as communication by sharing an object allocated in the process memory (RAM). Simple is included to exclude inter process Shared Memory Communication, like Remote Direct Memory Access (RDMA).

As the company grew, the supplied functionality also grew, and the architecture became more complex (See Figure 2).

The Locks have been extended with a number of external devices, using the Locks as a communication gateway or interacting directly with them. These devices communicate on a number of wired and wireless standards, e.g. ZigBee, BLE, RS-485, 869MHz.

The Control Service has also been extended to integrate third party systems, including emergency call systems, Smartphone applications and integration with external user databases.

Currently we experience on average once a week a situation where we have so many timeouts from communicating with Client REST Services that Performance is negatively affected. Over the last 6 months we have experienced twice that the timeouts in communicating with Client REST Services has crashed the Control Service (0 Availability).
The key differences are:

1. The use of the Lock as a communication gateway for sensors and actuators requires the protocol to be extended, and the Control Service understanding these new types.
2. When the lock functions as a communication gateway, the data is often piped directly to an external server, often via a REST API and these communication gateways have a much higher message frequency. Where the average lock communicates maybe 10-50 event messages and 24 heartbeat messages a day, the communication gateways can easily send 100,000 messages a day.
3. The use of many external services and dependents, whether Client REST receivers, booking systems, or other, has introduced a new risk to Availability, as these external services, if unavailable or slow, introduce a significant load to the service, and may cause a cascading failure.
4. The continuous introduction of new Server Applications, sensors and actuators on the locks and other supported HW/protocols, puts a heavy strain on the Control Service, as every modification requires the service be redeployed, causing all Locks to reconnect, which is undesired as mentioned above and introduces a higher risk of failure.
   a. Server Applications often communicate with the REST service. This is to avoid rewriting the Control Service whenever possible, as the REST service exposes a relatively generic interface. The use of the REST service does however introduce an unnecessary complexity and authentication requirements, which is unnecessary when operating internally (on same server or secure network).

The new dependency on external services as well as the increase in requests for custom solutions by clients has increased the need for Modifiability dramatically.
Please refer to Appendix B – Communication choices for more information about the internal and external communication choices.

2 Problem formulation

In this section will first be presented a set of scenarios, which sets the context for the hypothesis starting in section 2.3. Each scenario is used to validate or disprove the corresponding hypothesis.

2.1 Scenario 1 – Client REST service becomes non-response

All transmission of messages to REST receivers are handled in a thread pool. A queue is designed to ensure that the transmissions are grouped, though naturally the queue timeout must be small to avoid unnecessary delays in communication, currently set to between 2 and 10 seconds. The transmission to the REST Client receiver is done asynchronously, and so all transmissions in the group are transmitted in parallel. Unfortunately, if a Client REST service becomes non-responsive (or very slow), then the transmission will be tied up until a response is received or a timeout. Some Client REST services receive several messages every queue timeout, meaning that many pending transmissions may be spawned before the first Timeout occurs. If the Client REST service remains non-responsive or slow it may exhaust the transmitters pending queue thereby severely reducing Performance or even threatening Availability, in some cases crashing the Control Service thereby triggering a service restart with unwanted Lock reconnects as a result. This can be found as a Nygaard antipattern called dogpile in the second edition of Release It! [9].

![Diagram](Figure 3 – Non-responsive Client REST service)

2.2 Scenario 2 – Control Service needs to have extended functionality

The Control Service has many dependencies, and when it needs to be updated, the dependencies experience loss of availability, as well as the Locks must reconnect and exchange codes and status information.
The Control Service consists of many assemblies and namespaces. The external communication channels are separated into assemblies so they may be shared. The External communication is always decoupled from the Control Service by at least Inter Process Communication (IPC), though both internal and external may be executing on the same server. Internally the responsibilities are separated into namespaces, and in Figure 5 the overall separation for the Locks and the REST Service may be seen. The Business logic consists of many namespaces and some sub-assemblies and is used to indicate all other functionality than the Lock receiver and the IPC receiver. The internal communication in the Control Service is based on synchronization of shared objects in memory. This allows for very fast communication, but also impose Common coupling [6]. The term Common coupling is used to indicate that communication is done through shared data, and this is not to be mistaken for Semantic coupling or Content coupling.
Of the many possible functionalities which may be added to the Control Service, this scenario deals with one specific often occurring functionality, which is also representative for almost all changes, as the change affects the Locks, the Control Service and the REST service.

When a new firmware is developed and deployed in a Lock, a new Shared type may need to be created. The types are kept backwards compatible, but the new type must be known by the Control Service to be able to create REST or DB listeners (or other listeners for that matter). If the REST service is to be able to understand this new type, it must also be updated (only a subset of listeners is supported on the REST interface). The communication between Locks and Lock receiver is loosely typed (header + binary data), and it is the Business logic which parses the received data. The IPC receiver is strongly typed and requires access to the type definition to be able to process requests for the new type.

Naturally for a Client REST service or Website Control Panel to be able to receive or process the new type their Shared types must also be updated, but by keeping backwards compatibility in these Shared types as well, this can be done on request and only when the end user needs to process the new type. This update is irrelevant for the Control Service.

2.3 Hypothesis 1

Using Nygaard’s Circuit Breaker and Timeout Stability Patterns [5] to prevent the Control Service Quality of Service (QoS) to drop by more than 5% when the Client REST service becomes non-responsive, and without increasing resource consumption by more than 5%. This hypothesis considers scenario 1.

2.4 Hypothesis 2

Using the Bass et al Defer Binding Time and Localize Modifications Modifiability Tactics [2] to allow replacing the Business Logic at runtime, while keeping the locks connected and without increasing resource consumption by more than 5% or reducing the QoS by more than 5%. This hypothesis considers scenario 2.
3 Method

To validate the hypotheses, it is required to define the QoS and resources of the Control Service.

The Control Service QoS, as it relates to this report, is concerned with Availability. Availability is measured by looking at the responsiveness of the Control Service when changes occur.

As the primary functionality of the system is as an access control system, the propagation of access code changes is a vital functionality. There are many other metrics one could look at, but the propagation of access code changes involves the key components of the Control Service as well as the database. For this reason, the propagation of access code changes is sufficient to evaluate availability. The propagation of access code changes is defined as the time from a user code is requested changed, until it is sent to the Lock, as may be seen in Figure 6. In this report the metrics for the Control Service QoS will be referred to as availability.

![Figure 6 – User code change propagation](image)

Resource consumption, as it relates to this report, is concerned with the consumption of computing resources by the Control Service. In this report the metrics for these computing resources will be referred to as performance values.

The performance values of interest o this report is

- CPU time
- Memory
- Thread count

Network utilization has been excluded as it is not relevant to the scenarios related to the hypothesis.

Only the external dependencies mentioned in the scenario will be considered, all others will be disabled during the test.

UML will be used to document any solution designs.
3.1 Hypothesis 1

To validate the first hypothesis a scenario representing a realistic load situation must be created. As the hypothesis relates to communication with the external REST servers, this is done by looking at the real-life scenario with the highest number of calls to external REST servers.

The performance and availability parameters will be recorded during execution of the test scenario. As the Control Service fluctuates a lot in especially CPU usage, comparing performance requires measuring over a period and recording the average and peak values.

For the CPU usage the method for recording will rely on processor time allocated to the Control Service process. Memory looks at the total memory allocated to the Control Service process. The total memory is defined as the private memory as well as the shared memory in libraries or operating system – In Windows this is referred to as Virtual Memory. The thread count is the number of threads allocated to the Control Service process.

To record the average and peak values the performance parameters must be sampled over a period. Internally the Control Service has a 2 second timer in the message queue, and therefore a value larger than 2 seconds should be used. For this reason, the performance parameters will be sampled every second and accumulated over 30 seconds before peak and average is logged.

Availability cannot be measured without affecting the Control Service, and therefore measuring availability will not be performed during the measurement of performance, but rather before and after.

To determine the behaviour of the Control Service and the tested solution under different loads, tests will be performed with a different number of messages being transmitted.

3.2 Hypothesis 2

To validate the second hypothesis a scenario representing a realistic load situation must be created. As the hypothesis relates to communication between the locks and the Control Service, this is done by looking at the real-life scenario with the highest number of events from locks.

The same method as used for Hypothesis 1 may be employed here. During the Control Service update the availability will be 0 and comparing availability during the update would not make sense, and the availability is ignored in this period. The Control Service update period is considered under Customer QoS and is not part of this report.

3.3 Deployment

- Windows server 2012, 1 CPU 8GB RAM
- MySQL Database, 4GB memory allocated for caching and indexing.
- WCF for REST services
- IIS 7.x and ASP.NET
- Control Service and other services written in C# .NET 2 and 4

Please refer to Appendix A – Realistic message activity for more details about the messages generated.
3.4 Definitions

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client</td>
<td>The end user or the organisation the end user has employed to represent him or her, e.g. a service organisation maintaining the gate is responsible for receiving and handling SMS and E-Mail notifications.</td>
</tr>
<tr>
<td>~</td>
<td>Approximately. Indicates that the number is not exact, but rather a rounding of a calculated number, but where including all significant figures would give a false indication of accuracy.</td>
</tr>
<tr>
<td>Lock</td>
<td>A HW device with a GSM modem (optionally WiFi module) continuously connected to the Control Service and optionally external sensors via a number of RF modules. This is in code often referred to as SBC (Single Board Computer), and in this report Lock and SBC should be seen as synonyms.</td>
</tr>
<tr>
<td>CS (Control Service)</td>
<td>The service responsible for all communication with the Locks and all business logic relating to the Lock and interested services and Clients. Control Service is abbreviated CS.</td>
</tr>
<tr>
<td>Event</td>
<td>When used in the context of being sent from a Lock, it specifies an unsolicited message as opposed to a response to a command from the server. Often events are triggered by something external to the Lock, like a pin code being entered, a sensor sending data to the lock, etc.</td>
</tr>
<tr>
<td>Access code</td>
<td>Access granting data registered with a user in the database and copied to all Locks which the user has permission to unlock. An Access code may be e.g. an RFID, a pin code, a phone number, etc.</td>
</tr>
<tr>
<td>Impulse</td>
<td>Defined by Nygaard[5] as a rapid shock to the system. This could for this system be a sudden increase in the number of messages which must be transmitted. This can occur if e.g. the phone company experience a national outage. When the connection is restored all locks will connect and transmit their cached messages at the same time.</td>
</tr>
<tr>
<td>Stress</td>
<td>Defined by Nygaard[5] as the continuous state of high load due to lack of resources or bad design. This could for this system be seen when availability drops due to lack of RAM for the database cache, or in-proper indexing. Or simply the design of the database being sub-optimal as the number of queries increase.</td>
</tr>
<tr>
<td>Strain</td>
<td>Defined by Nygaard[5] as the change in behaviour in the stressed system or dependent systems. When a system is under stress, this stress (delay) will propagate to other dependent systems, causing stress on those systems or other unexpected behavior.</td>
</tr>
<tr>
<td>Catastrophic failure</td>
<td>The system or application having 0 Availability in vital functionality, and do not recover without the system or application being restarted.</td>
</tr>
</tbody>
</table>

In this report, the number and date format will be Danish, meaning the ‘.’ will be used as a thousand separator and the ‘,’ as a decimal indicator. Dates will be expressed as dd-MM-yyyy.

4 Tools

In order to evaluate the hypothesis some tools and test stubs must be employed or developed.

All tools are written by the author expressly for the purpose of this report, with the exception of the Lock simulator (4.1), which was written by the author prior to this report, and was augmented with functionality to create and use message transmission scenarios.

4.1 Lock Simulator

The Lock simulator is based on a pre-existing Lock simulator written to simulate the behaviour of the locks. The pre-existing lock simulator is used extensively during testing of new Control Service functionality or to test how the Control Service handles new lock messages or behaviour. The Lock simulator is continuously updated to
behave like the actual lock firmware, yet naturally with many more possibilities for configuring incorrect behaviour, like TCP-disconnect, drop (no disconnect sent), no TCP-ACK on command, no response on command/heartbeat, invalid message format, TCP-ACK response delay, command response delay, etc.

The Lock Simulator is named SBC Simulator for historical reasons where the Locks mainboard was named SBC (Single Board Computer). Therefore, the classes within the simulator relating to the Lock is named SBC.

For the purpose of this report the pre-existing Lock simulator needed to be augmented with new functionality.

1. Configure a number of simulated locks to continuously transmit events according to a schedule
2. Generate an event transmission schedule from a recording of actual lock behaviour. The recording of actual lock behaviour may be found in the existing service log and communication log files.
   a. The heartbeat message between the Control Service and the locks, which is not included in the service log, may be configured to execute periodically, but due to the relatively short testing period it is not relevant.

![Figure 7 - Lock simulator eventing class diagram](image-url)
The simulator uses as input a list of lock ids to simulate along with information about the location (Endpoint) of the Control Service. Additional configuration parameters may be added to control the specific behaviour of the lock simulation, yet for the purpose of this report it is not necessary.

Once the simulated locks are connected the Event View may be used to add and send messages to the Control Service.

Please refer to Appendix C – Tool description for some screenshots of the tool and its functionality.

### 4.2 Client REST test double

To setup and control a number of REST message receivers, it is possible to look for an existing implementation, like Mountebank [8], or develop a custom version.

Developing a custom version should only be done if it is necessary (or simpler).

This project will use the Mountebank test double to create a number of REST receivers, which can be configured with a given delay. The version used is version 1.14.1 for windows 64bit.
There are many aspects to configuring a REST server and client. For some background knowledge about the intricacies please refer to Appendix I – REST communication considerations.

4.3 Performance and Availability evaluation tool (PAAET)

To evaluate the hypothesis the performance and availability must be evaluated.

The availability evaluation is specific to the Control Service, so it is necessary to build a custom tool to evaluate this.

Performance may be evaluated using an existing tool, e.g. Apache JMeter.

As a tool has to be written anyway, and as the Diagnostics library in .NET allows retrieving the required information quite simply, it has been decided to integrate the performance measurement into the tool. This is done by evaluating the learning curve of the alternative (Apache JMeter was considered and compared) along with the complexity of adding it to the tool, and it was found to be more difficult to use JMeter.

Had a more advanced measurement been required then most likely an existing tool like JMeter would have been beneficial.

The tool adheres to the method described in the Method section.

To measure access code change propagation the tool uses the simulated lock instance reversely. The evaluation tool uses the Control Service IPC API to trigger a change in an access code which must be transmitted to the registered evaluation tool lock simulator and measures the time until the access code change reaches the Lock simulator instance, as may be seen in Figure 9.

![Figure 9 – Evaluation of Control Service availability](image-url)
Performance is measured by simply sampling the Control Service process for CPU time, memory and thread count every second for the 30 seconds, as specified under method.

Please refer to Appendix C – Tool description for some screenshots of the tool and its functionality.

5 Analyses and Results

This section details the individual scenarios, the implementation of a solution and the evaluation of the hypothesis.

Each scenario is handled independently following this plan:

1. Describe the scenario in detail, solutions and considerations
2. Document and describe the solution
3. Test data
4. Execute test runs
5. Evaluate hypothesis

5.1 Scenario 1 – Client REST service becomes non-response

As keeping the Client REST service responsive is the responsibility of the Client it is completely acceptable to simply blacklist the given endpoint for a time when its responsiveness drops under a predefined value.

The Nygaard Circuit Breaker pattern [5] is very well suited for just this situation. The Nygaard Circuit Breaker pattern is, as the name implies, inspired by the electrical circuit breaker. The principle is that if the dependency is not behaving properly then don’t use it. An electrical circuit breaker has a value at which it trip, and the same is true for the Circuit Breaker pattern. Unlike the electrical circuit breaker, a remote endpoint may become responsive again “by itself”. Perhaps a third party was causing high load on the remote endpoint and has now stopped. For this reason, it makes sense to automatically open the architectural circuit breaker and try again periodically.

A remote endpoint may behave inappropriately in different ways.

- **Respond slowly or no response at all**
  
  If the remote endpoint starts responding slowly and if more calls to this endpoint is performed then it can process, then the number of pending calls will grow indefinitely and eventually consume all resources of the caller. Therefore, the circuit breaker is designed to only permit a certain number of simultaneous delayed calls.

  A circuit breaker permitting only a certain number of ongoing calls – ongoing, not delayed, would also function. The reason the circuit breaker deals with delayed calls instead of ongoing here is because a sudden burst of simultaneous calls is expected and occur now and again. If the remote endpoint is functioning then it will process the calls and quickly lower the number of ongoing calls, preventing them from ever becoming delayed.

  However, if there are many delayed calls, then it is an issue on the remote endpoint, which may cause a cascading error. However, the total number of simultaneous calls will be addressed below under Bulk Heads considerations.
• **Refuse connection or respond incorrectly**

If a remote endpoint responds incorrectly, i.e. if a REST service specified to return 200 OK, returns e.g. 400 NOT FOUND, then it is highly unlikely that subsequent calls to the same service will succeed, and the circuit breaker may therefore trip quickly on incorrect responses, and if connection is refused then even more so.

As mentioned a short spike in the number of calls to a functioning service may not be a problem. Perhaps the service will be a little slower for a short while as it processes the calls, but if the Control Service generates a very large number of calls, perhaps over a longer period, this could be a problem. Not only will it put a strain on the Control Service, it may also cause the called service to fail.

In order to protect against a part of a service running amok and consuming unacceptable amounts of resources, one can use another pattern, e.g. the Nygaard Bulk Heads pattern. A Bulk head is a nautical term used to describe sections in a ship which can be hermetically separated from each other. This means that if one section of the ship is breached and is flooded with sea water, then the separation into hermetic sections prevent the entire ship from filling with water, and thereby keeping it afloat. This was indeed the idea behind the Titanic, and if only a few sections had been breached by the iceberg, it would most likely have worked.

Preventing the notification part of the Control Service (the part performing the calls to the external services) from consuming unlimited resources, can help protect against a software error or unexpected external behaviour causing a cascading error, making the entire Control Service fail.

A way to design the Bulk heads pattern in this case could be to integrate it into the circuit breaker design. Just like the circuit breaker monitors the time used in calls to external services and the number of delayed calls in progress, the circuit breaker could also monitor the total number of calls to a single endpoint or in total, thereby preventing the Control Service notification part from scheduling an unacceptably high number of calls.

This is not truly a part of the circuit breaker pattern, as it relates to the calling part as opposed to the called part, and so the design becomes a combination of circuit breaker and bulk heads pattern.

### 5.1.1 Pattern design and implementation

To design a circuit breaker as described above and use it in the existing Control Service, it is beneficial for the design to be easily integrated into the Control Service, and also allow easy testing.

As the design is a protection mechanism which will not be relevant 99,9% of the time, it is important to ensure that the performance cost is minimal when not triggered.

The current implementation is designed to use asynchronous transmissions, meaning the method will return immediately and then perform a call-back in a different thread when the execution is done, regardless of whether it succeeds or fails, so naturally the circuit breaker must also support this.

```csharp
foreach (var v in sendRestMsgs.GroupBy(x => x.Addresses))
{
    RestMsg[] rm = v.ToArray();
    MonitoredThreadPool.QueueUserWorkItem(delegate
    {
        foreach (RestMsg sm in rm)
        {
            RestMsg localSm = sm;
            _restTransmitter.SendMessageAsync(sm.Payload, sm.Addresses, delegate(bool b)
            {
                if (!b)
                {
```
Furthermore, it is necessary that a circuit breaker only affect the given endpoint, so a tripped circuit breaker do not stop all calls to REST clients. This is done by having the design contain a number of circuit breakers, one for each endpoint address. To support the use of the circuit breaker for other needs, e.g. an Email transmission gateway, SMS gateway, or any other dependency which may become non-responsive and should then be turned off.

In Figure 10 can be seen the class diagram for the SlaCircuitBreaker and its dependencies.

As it may be seen it differentiate between synchronous and asynchronous calls, which is required to know when execution is done. Asynchronous calls require a call to EndAsyncExecute where the synchronous Execute can be considered done when the call-back returns.

As mentioned each REST client endpoint has its own circuit breaker, and this is achieved by using an id to identify the circuit breaker to consider, and each id represents an instance of the SlaCircuitBreakerEntry, which in turn represents a circuit breaker. Each SlaCircuitBreakerEntry maintains a state machine, as may be seen in Figure 11.
An example of the use of the SlaCircuitBreaker can be seen in the sequence diagram in Figure 12. Here the most common scenarios can be seen. The design has been made in such a way that it is unnecessary to have a dedicated thread for the SlaCircuitBreaker, and the functionality is achieved by evaluating the state machine before calling the application action.

Apart from the indicated design a safety mechanism is introduced to limit the effects of a SW bug in the application code. If e.g. the asynchronous initiated by the application in the SlaCircuitBreaker call-back never finishes (no call ended), then that execution will remain indefinitely in the SlaCircuitBreakerEntry, and if enough of these exist will keep the circuit breaker open indefinitely. For this reason, a maximum execution time may be set, and any actions not completed in this time will be considered stuck and are removed from the list. An error is generated if this happens to allow debugging of the problem. Naturally events are also triggered in case of a circuit breaker trip, so the maintenance team can evaluate the external dependency.
foreach (var v in sendRestMsgs.GroupBy(x => x.Addresses))
{
    RestMsg[] rm = v.ToArray();
    MonitoredThreadPool.QueueUserWorkItem(delegate
    {
        foreach (RestMsg sm in rm)
        {
            RestMsg localSm = sm;
            _scb.ExecuteAsync(sm.Addresses, delegate (SlaCircuitBreakerResult result)
            {
                _restTransmitter.SendMessageAsync(localSm.Payload,
                        localSm.Addresses,
                        delegate (bool b)
                        {
                            _scb.EndAsyncExecute(result.Id, b);
                            If (!b)
To test the implementation before deployment a set of UnitTest has been implemented, taking advantage of the fact that the SlaCircuitBreaker is designed with not only testability in mind, but also to protect any synchronous or asynchronous execution, and so the test is quite easy to write. See Appendix H – UnitTest and results and - Appendix D – Test and configuration data for details about the test.

5.1.2 Test data

Given the use of the system it is known where the main concentration of events. Based on that, it is known that the main concentration of messages is in the morning when everyone gets up or in the evening when they are put to bed. Morning routine is more precise than the evening, and therefore has the highest concentration of messages over a small period of time. Choosing a recording around 8AM on the 4th of April 2018 shows that the REST transmitter sent 4086 events from 7:45 to 8:15, or 2.27 events per second on average. On top of this comes a smaller number of events written to the database, but not sent to any REST client. This number is insignificant, and can therefore be ignored, simplifying the test.

Internally the Control Service processes each message against cached data in RAM in highly efficient dictionaries, and all messages types has a similar load on the Control Service, and the choice of message type is therefore irrelevant. The same is true for the origin of the message (the specific lock).

The database, lock simulator and REST client test double is configured to perform the following test.

1. Send N messages in total a second
   a. Two different message types
   b. From two different lock simulator instances; lock1 and lock2

2. Process the messages and send:
   a. Message type 1 send to REST client 1 and REST client 3
   b. Message type 2 send to REST client 2 and REST client 3

The pauses between the lock simulator sending messages has been chosen to differ slightly, yet remain under 2 seconds, which is the Control Service queue processing interval (how often the message transmission and logging is processed).

The value of N is chosen to be 3, 15, and 75, to test under different load scenarios from normal heavy load and increasing by a factor of 5.

The Control Service database is generated using an SQL script, as shown in Appendix D – Test and configuration data. The script generates a database with:

- A single customer
- Y locks + 1 for the performance tool
• Each lock contains Z access codes.
• 1 database log listeners (events to be written to the database log)
  o New lock discovered event
• 3 notification listeners (REST Client 1 – RESTR client 3) matching the Client REST Simulator listening to
  the events as described above

The value of Y is set to either a very limited number of units (Y = 2) with no access codes (Z = 0) or a realistic
number of locks (Y = 5000) with a realistic average number of access codes (z = 100).

The Control Service is configured for four different test runs:
1. Without the SLACircuitBreaker and with a REST timeout of 300 seconds
2. With the SLACircuitBreaker and with a REST timeout of 300 seconds
3. Without the SLACircuitBreaker and with a REST timeout of 25 seconds
4. With the SLACircuitBreaker and with a REST timeout of 25 seconds

The lower REST timeout shows the behavior of the Control Service if the REST call times out quicker, thereby
limiting the number of ongoing calls, and also lower the testing time required.

These tests will be used to show and verify the pattern in the availability and performance, thereby validating or
dismissing the hypothesis.

All setup-files may be found in Appendix D – Test and configuration data.

5.1.3 Test execution
The testing procedure may be seen below. Step 2 – 6 may be done using the prepare_setup.bat file and the
desired run_N_type.bat file from the execution directory. Please refer to Appendix D – Test and configuration
data for details about the test setup.

1. Start mountebank
2. Initialize DB
3. Prepare and start Control Service
4. Prepare and start Evaluation tool and
5. Configure mountebank to have the three REST listeners.
6. Start Lock simulator
7. Connect locks from lock simulator
8. Begin transmission of messages from lock simulator
9. Introduce delay of 10s on single receiver
10. Change delay to 30 seconds on that receiver
11. [Change delay to 330 seconds on that receiver]

Between each step a delay is inserted to allow a steady-state to be reached in the Control Service, which
simplifies comparison.

The test is executed for all three message transmission frequencies as well as with a small number of units and a
realistic number of units. The delay of 330 seconds is only tested for a REST Timeout of 300 second.

When the SLA circuit breaker is enabled it is done with the following parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
</table>

Aarhus University | Department of Computer Science
IT-Vest | Master of IT
RestSlaCircuitBreakerEnabled | true | Indicate that the SLACircuitBreaker should be used. If false or omitted use original implementation.
--- | --- | ---
RestSlaMaxDelayMs | 5000 | The maximum amount of time permitted to complete a given REST notification before the notification is considered delayed.
RestSlaMaxDelayedExecutions | 5 | The maximum number of delayed REST notifications before the SLA Circuit breaker trips.
RestSlaMaxExecutions | 2000 | The maximum number of ongoing executions permitted, delayed or not, before the SLA circuit breaker trips.
RestSlaCircuitBreakerDelayS | 60 | The amount of time the circuit breaker is tripped before it becomes semi-open and tries a single notification.
RestSlaMaxFailure | 3 | The number of failed notification responses before the SLA circuit breaker trips.
RestSlaMaxExecutionTime | 3600 | A failsafe parameter protecting against stuck executions. Any ongoing execution older than this value will automatically be cleared from the pending queue.

The above values are chosen by considering the practical use. The clients receiving many requests are required to have a receiving host with high performance and availability, thereby ensuring fast processing, whereas the customers receiving occasional notifications, may have a receiving host with much lower specifications. As the configuration permits up to 5 delayed executions, this will solve any issue with a host receiving occasional notifications and spending more than 5 seconds processing them. At the same time the 5 second value allows the SLA circuit breaker to react relatively quickly if a host receiving many requests becomes unresponsive.

**Results**

The results of the test can be summarized in Table 3. The table shows the difference in Availability and the Performance parameters, when the REST receiver is functioning as expected. Negative values indicate that the test with the SLA circuit breaker enabled has a lower value than the version without the SLA circuit breaker enabled.

<table>
<thead>
<tr>
<th>Difference in % at no delay</th>
<th>Availability</th>
<th>CPU</th>
<th>Memory</th>
<th>Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 locks 3 msg/s</td>
<td>-2.48</td>
<td>-2.08</td>
<td>0.43</td>
<td>5.17</td>
</tr>
<tr>
<td>2 locks 15 msg/s</td>
<td>4.71</td>
<td>-0.38</td>
<td>-0.06</td>
<td>1.61</td>
</tr>
<tr>
<td>2 locks 75 msg/s</td>
<td>-2.54</td>
<td>2.64</td>
<td>0.03</td>
<td>1.41</td>
</tr>
<tr>
<td>5000 locks 3 msg/s</td>
<td>-2.64</td>
<td>-1.91</td>
<td>0.64</td>
<td>-5.17</td>
</tr>
<tr>
<td>5000 locks 15 msg/s</td>
<td>-0.22</td>
<td>-0.67</td>
<td>-0.19</td>
<td>-1.56</td>
</tr>
<tr>
<td>5000 locks 75 msg/s</td>
<td>2.48</td>
<td>-2.53</td>
<td>2.14</td>
<td>4.00</td>
</tr>
<tr>
<td>Average</td>
<td>-0.12</td>
<td>-0.82</td>
<td>0.50</td>
<td>0.91</td>
</tr>
</tbody>
</table>

Table 3 - Scenario 1 results difference no delay

Looking at the average of all measurements the difference is well within the 5% on the performance parameters as well as the Availability.

As the measurement is done on a process executing in a managed environment and employing a thread pool, and a garbage collector, it is expected that there will be some deviations because the thread pool spawns and closes threads as needed and the garbage collector frees memory on its own accord. Furthermore, as the test is performed on a server which executes other programs, both dependencies like the database and others, as well as the operating system, some deviation can also come from other server activities. The total load on the server has been kept to a minimum and all non-essential processes has been closed, so the deviations from other processes should be at a minimum.
For these reasons, multiple measurements are used and the average is then compared. Even with this averaging some deviation is observed. Especially in the low message number tests. This is due to the low message tests having a much lower general load, causing especially the CPU load and thread count to be low. If the CPU load and thread count is low, then even a small change will result in a large percentage difference.

In Table 4 may be seen the same comparison except with the REST timeout set to cause a timeout.

<table>
<thead>
<tr>
<th>Difference in % at max delay</th>
<th>Availability</th>
<th>CPU</th>
<th>Memory</th>
<th>Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 locks 3 msg/s</td>
<td>-3,25</td>
<td>-47,67</td>
<td>-6,89</td>
<td>-7,69</td>
</tr>
<tr>
<td>2 locks 15 msg/s</td>
<td>-41,61</td>
<td>-27,63</td>
<td>-7,38</td>
<td>-6,25</td>
</tr>
<tr>
<td>2 locks 75 msg/s</td>
<td>-53,89</td>
<td>-41,54</td>
<td>-8,38</td>
<td>-16,13</td>
</tr>
<tr>
<td>5000 locks 3 msg/s</td>
<td>-8,05</td>
<td>-40,00</td>
<td>-9,84</td>
<td>-14,58</td>
</tr>
<tr>
<td>5000 locks 15 msg/s</td>
<td>-7,96</td>
<td>-35,47</td>
<td>-4,79</td>
<td>-7,37</td>
</tr>
<tr>
<td>5000 locks 75 msg/s</td>
<td>-13,72</td>
<td>-27,31</td>
<td>-11,19</td>
<td>-17,20</td>
</tr>
<tr>
<td>Average</td>
<td>-21,41</td>
<td>-36,60</td>
<td>-8,08</td>
<td>-11,54</td>
</tr>
</tbody>
</table>

Table 4 - Scenario 1 results difference max delay

As it may be seen the effect of the SLA circuit breaker increase as the number of messages increase, which is due to the increased number of messages that the SLA circuit breaker prevents from being sent and thereby delayed.

Looking at the actual values is it possible to see that the availability and performance is almost unchanged when the REST timeout is increased, indicating that the SLA circuit breaker is able to prevent the resource consumption to increase as well as the availability from deteriorating. This is shown in Table 5.

<table>
<thead>
<tr>
<th>2 locks 15 msg/s</th>
<th>Availability [ms]</th>
<th>CPU [%]</th>
<th>Memory [MB]</th>
<th>Threads [count]</th>
</tr>
</thead>
<tbody>
<tr>
<td>No delay</td>
<td>75,66</td>
<td>2,46</td>
<td>232,17</td>
<td>21,00</td>
</tr>
<tr>
<td>10s delay</td>
<td>75,34</td>
<td>2,07</td>
<td>232,91</td>
<td>21,33</td>
</tr>
<tr>
<td>30s delay</td>
<td>76,33</td>
<td>1,74</td>
<td>231,99</td>
<td>20,00</td>
</tr>
</tbody>
</table>

Table 5 - Scenario 1 results SLA circuit breaker stability

2 locks with 15msg/s was chosen as it is enough messages to give a clear indication that the SLA circuit breaker is working, but not so may that the general message handling begins to affect performance.

This can be compared with the values for the test without the SLA circuit breaker enabled, as shown in Table 6.

<table>
<thead>
<tr>
<th>2 locks 15 msg/s</th>
<th>Availability [ms]</th>
<th>CPU [%]</th>
<th>Memory [MB]</th>
<th>Threads [count]</th>
</tr>
</thead>
<tbody>
<tr>
<td>No delay</td>
<td>72,26</td>
<td>2,47</td>
<td>232,30</td>
<td>20,67</td>
</tr>
<tr>
<td>10s delay</td>
<td>85,74</td>
<td>2,63</td>
<td>236,68</td>
<td>23,67</td>
</tr>
<tr>
<td>30s delay</td>
<td>130,74</td>
<td>2,41</td>
<td>250,47</td>
<td>21,33</td>
</tr>
</tbody>
</table>

Table 6 - Scenario 1 results no SLA circuit breaker

The comparison of availability between the tests with SLA circuit breaker and the ones without may furthermore be seen in Figure 13.
Finally, when the SLA Circuit Breaker is active, analyzing the mountebank log files (and/or console output), gives information about the behavior of the SLA Circuit Breaker. It is possible to see that the REST endpoints not set up to impose a delay, continue to receive events. The endpoint with a delay only receive the semi-open attempt according to the SLA Circuit Breaker configuration. When the delay is removed and the semi-open attempt succeed, then all endpoint receives continuous events.

All the results may be seen in Appendix D – Test and configuration data and some graphs with interesting observations may be seen in Appendix J – Selected graphs.

5.1.4 Evaluation

In the test execution it can be seen that the SLA Circuit Breaker prevents the large increase in resource and reduction in availability, which is observed when a REST receiver becomes non-responsive. This proves that the SLA Circuit breaker does indeed alleviate the resource issue from the slow or non-responsive REST receiver while maintaining the QoS requirements.

More fluctuation in the results was observed than expected. A lot of this can be referred to influences outside of test control, like the operating system, the database service, the executing framework, etc. Gaining test control over many of these influences is possible but is outside the scope of this report. Instead heavy use of averaging has been employed to limit the effects of these rogue values.

The strong increase in memory use and drop in availability when the REST transmitter begins to time out was also not expected. Evaluation of the tests and analysing the code as well as the lower implementation indicates that the underlying implementation consumes many resource and impose delays when many asynchronous REST transmissions begins to time out. This further indicates the importance of the principle of “if it hurts, don’t do it”.

The performance requirements in the hypothesis requires that the implementation do not impose more than 5% extra resource consumption. By comparing the resource consumption with no delay in the REST receiver, it is possible to see that this is met with room to spare.
Based on this it is believed that the hypothesis has been proven.

Further testing of the exact values of the REST transmitter timeout and the SLA Circuit Breaker configuration should be performed to find the optimal values.

5.2 Scenario 2 – Runtime service update

In Bass et al. the Modifiability is split into Localize modifications, Prevent Ripple Effects and Defer binding time. Localize modifications and Prevent ripple effect are very similar to the concept of high cohesion and low coupling. On a class level it is good to avoid dependencies on internal structures (coupling) as well as keeping the class well defined and with a clear responsibility (cohesion). This principle also translates to libraries and processes.

Using shared libraries are vital in creating maintainable code. It allows for code reuse and for not only gaining the improvements to the library on all users, but also allowing for a bug to only need fixing in one place. The downside of shared libraries is that if the library changes its interface then it may break the processes which has not been updated to the new interface. This is why backwards compatibility is very important.

Keeping a high cohesion in one’s libraries makes it simpler to maintain an overview of where a changed library may have an effect. It is always confusing when using a library causes a dependency to an entirely different and logically unrelated library, because someone perhaps had a shared class and needed somewhere to put it, and therefore chose a shared library, regardless of the intended function of this library.

The component-connector diagram can be very useful in getting an overview and identifying strange dependencies.

When a library needs to be updated naturally all dependencies must be updated to use this new library. For this reason, limiting the number of dependencies improves modifiability. Limiting the number of dependencies relates to low coupling, in that the fewer dependencies the lower the coupling.

These two points of using shared libraries to gain code reuse and not using shared libraries to improve modifiability are somewhat conflicting, and a balance must be achieved. By focusing on responsibilities, it is possible, at design time, to consider where to place the different functionalities and at what abstraction level. By grouping responsibilities according to not only the design, but also the developer’s past knowledge of the architecture and previous changes, it can be attempted to obtain an architecture where individual changes are localized to as few libraries as possible, while still achieving a high level of code sharing.

As mentioned above the abstraction layer of the library is also important. If two processes share a type from a library, and that type is changed to accommodate one process, this may break the other process (if backwards compatibility is not maintained). Maintaining backwards compatibility at a typed level can be complicated and can lead to a lot of interfaces which exist only to maintain backwards compatibility.

An alternative is to use a more general type. If, instead of a specific type, the two processes use a serialized version of the type (e.g. byte-array or string (JSON)) when communicating with the library, then the library can be updated to handle more complex types without changing the library, as the library simply evaluate the received and determine what version it is (e.g. which data fields it has).

Again, a balance must be achieved. On one side using shared types, which improves readability and compile time enforcement of required types, and on the other using more general types, which improves modifiability but relies on runtime testing for type safety.
Replacing a library can be done in more than one way. The simplest is naturally to just replace the library and re-compile/re-spawn the process. This may not always be beneficial, and then the Defer binding time tactic comes in handy. If the library which the process is dependent upon is written with a well-defined and general interface it is possible to re-bind the library at runtime, though updating type definitions in use is not for the faint of heart.

An alternative is to move the libraries functionality into its own process and relying on inter process communication (IPC) to use the library. The observer pattern or publish/subscribe can be used to register callbacks if the library uses notifications. In this case IPC should be seen in its broadest sense, meaning any way for two processes to communicate, not just two processes on the same system.

This manoeuvre of creating a process for the library also forces the developer to consider the responsibilities and interface of the library as well as the abstraction level of communicating with the library. Naturally the cost is performance, as IPC communication will always be slower than using shared memory, which is possible if the library is compiled into the process.

The idea of having a program use external functionality in this manner is not a new one, and to the interested reader a short description about the history may be found in Appendix F – History of external functionality.

Due to the existing design of the Control Service, and the high complexity and risk involved with runtime type replacement, this option is discarded and will not be considered further in this report.

This leaves the separation of responsibility into separate processes.

Basically, scenario 2 involves moving the lock event queue and transmission stream from the Control Service to an external service.

This can be done by integrating e.g. the RabbitMQ message queue service or Kafka Event Stream service. RabbitMQ is a well-respected message broker, which allows it not only to serve as a message queue (MQ), but also to distribute (broker) the received messages to interested parties.

As the locks firmware is already in place and updating the lock firmware is outside the scope of this report, this would require inserting not only the RabbitMQ service but also a Lock protocol to RabbitMQ service, as may be seen in Figure 14.

This solution has the advantage of relying on a third party well tested message queue (MQ), making registering the Control Service to the MQ easy, and simplifying the further separation of the Control Service as RabbitMQ is a message broker, and can therefore distribute messages of interest to the correct specialized Control Service instance, depending on the Control Service registration to the RabbitMQ.
It also allows for new lock firmware being written to directly use RabbitMQ, though here a detailed analysis of the protocol data consumption must be performed, as the current proprietary protocol is designed for extremely small data amounts. This analysis is outside the scope of this report.

However, based on the hypothesis, a limit of maximum 5% performance penalty is impossible when introducing two new services/processes.

An alternative could be to use another Message Queue, e.g. MSMQ, for the IPC communication. As MSMQ is running already, it would not introduce as much of an extra penalty, and as the focus of scenario 2 is the Lock to Control Service communication, the broker functionality is not currently needed.

However, the MSMQ would require an extra piece of functionality. In RabbitMQ it is possible to know which connections exist at any given time (called channels in RabbitMQ). This is important when the Control Service is restarted without storing its state. In that case the Control Service must be able to get not only the events received while the service was down, but also get a list of currently active connections. This is not possible on a MSMQ based solution, and it would have to be implemented.

This may be done in several ways, one being Event Sourcing. Event Sourcing is an architectural pattern in which the state is determined by a sequence of events stored by appending only. In other words, by recording all events received by the message queue and evaluate them when the Control Service restarts it is possible to know the state of the Control Service when it was stopped. Same principle as a database log (transaction journal).

Another way is to place this state in the SBCConnectionRepository, which may then be queried when the Control Service starts.

Instead of modifying the Control Service it is also possible to utilize a TCP proxy. By placing a TCP proxy in front of the Control Service, the Control Service can function without modification. However, the proxy must function in a special manor, as it must be able to maintain the connection even if the Control Service is closed, which is not normal proxy behavior. It must also cache data received and connections received and lost when the Control Service is unavailable. This functionality is not available in COTS products at the time of this report, and it will therefore require developing a custom TCP proxy.

Based on the above a custom service/process must exist, henceforth referred to as SBCConnectionRepository. As this service must maintain a list of connections, there is no need to implement an Event Sourcing solution, as the service can simply expose the connections like RabbitMQ does.

The custom service could be implemented to use MSMQ, but as .NET Remoting IPC is already an integral part of the Control Service, and performance wise it is faster or the same, it has been chosen as a communication channel. By keeping the protocol dependent on only simple types, this communication channel can easily be replaced, e.g. by the RabbitMQ implementation when the need arises, yet this implementation is left for later works.

Attaching to the SBCConnectionRepository will be done using the same principles as with RabbitMQ (though greatly simplified), and fit very well with the Defer binding time principle of Bass et al.

As the chosen implementation utilizes IPC to separate the Control Service from the actual TCP connection the term IPC base Control Service or just IPC Control Service will be used to indicate the Control Service utilizing the separation of the Lock connections and the Control Service via IPC.
5.2.1 Pattern design and implementation

To allow the new implementation to be configurable it is required to change the existing design so the parts relying on the underlying communication layer (currently TCP) may be replaced.

The architecture already permits the communication stream to be replaced, as this is used in other projects. This is done by dependency injection where the underlying stream is wrapped in an implementation of the interface `ISBCProtocolStream`.

Unfortunately, the `SBCAcceptor` class, which is responsible for accepting new connections from locks relies directly on the `TCPLlistener` class, as may be seen in Figure 15.

![Figure 15 - Existing Lock acceptor design](image)

And an example of a lock connecting may be seen in Figure 16.

![Figure 16 - Existing Lock connection sequence diagram](image)
To allow for the TCPListener to be replaced with a version working over IPC a new interface, IConnectionListener, is designed to allow this. Again, dependency injection is used to choose between different implementations of the interface. An alternative could be to use the Factory pattern to have the SBCAcceptor class create the needed instance at runtime. However, it is known in advance that only a single instance of the IConnectionListener is needed, and it will exist for as long as the SBCAcceptor instance does. For this reason, the factory pattern would add an unnecessary complexity. The updated design with the new interface and the new implementation of the IPC version of ISBCProtocolStream may be seen in Figure 17.

![Figure 17 - Lock connection provider design](image)

In Figure 18 may be seen the same lock connect example that was shown in Figure 16. As it may be seen the only difference is the addition of the TCPConnectionListener, which performs the accept on behalf of the SBCAcceptor. This will not introduce a significant overhead, and the performance of this design will be equivalent to the original design, yet naturally the new design allows for the use of the IPC-based version. Other than simple code inspection this is further validated in the test data, where the original Control Service is included in the test.
Figure 18 - Standard Lock connection sequence diagram

An example of the lock connection example from Figure 18, yet using the IPCConnectionListener may be seen in Figure 19 and Figure 20. The figure has been split up into two to make the writing legible. The entire figure may be seen in Figure 30 in Appendix G – IPC single image diagram.
Figure 19 - IPC based lock connection sequence diagram (part 1)
To test the implementation before deployment a set of UnitTest has been implemented. Again, the consideration for testability allow for testing without the need for sleeps or external dependencies. A set of test stubs is designed to stub the TCPConnection and the SBCProtocolStream, and by using the Testability interface time and threads are stubbed away. The IPC layer is also removed (simplified) during unit testing by enabling special testing functionality on implementation via a testing interface (dependency injected into the executing code). See Appendix H – UnitTest and results and Appendix D – Test and configuration data for details about the test.

5.2.2 Test data

As the IPC based lock communication only involves the communication between the Control Service and the locks or the SBCLockConnetionRepository, the focus is on the number of lock connections and the communication.

For the number of locks and the message throughput, it is relevant to use the same parameters as used for scenario 1 as basis for the test data, yet as the external REST communication is irrelevant for this test, no REST
receivers are registered. Instead the messages, which were before sent to the REST receiver, is logged in the
database, to add some load and also to allow checking if any messages are lost.

The database and lock simulator are configured to perform the following test.

1. Send N messages in total a second
   a. Two different message types
   b. From two different lock simulator instances; lock1 and lock2
2. Process the messages and log them to the database

The value of N and the pauses between the lock simulator sending messages is chosen to be the same as for
scenario 1.

The Control Service database is generated using an SQL script, as shown in Appendix D – Test and configuration
data. The script generates a database with:

- A single customer
- 5000 locks + 1 for the performance tool
- Each lock contains 100 access codes.
- 3 database log listeners (events to be written to the database log)
  - New lock discovered event and the two events sent from the lock simulator

Experiments was performed with the 2 locks test, but they indicated that the sensitivity to external processes
and internal framework processing was simply too great, as the resource consumption was so small, and for this
reason these tests are not included in the report, though the test results may be seen in Appendix D – Test and
configuration data.

The Control Service is configured for two different test runs:
1. Not using the IPC interface
2. Using the IPC interface

These tests will be used to show and verify the availability and performance, thereby validating or dismissing the
hypothesis.

All setup-files may be found in Appendix D – Test and configuration data.

5.2.3 Test execution

The testing procedure may be seen below. Step 1 – 5 may be done using the prepare_setup.bat file and the
desired run_N_type.bat file from the execution directory. Please refer to Appendix D – Test and configuration
data for details about the test setup.

1. Initialize DB
2. Prepare and start the SBC Connection Repository Service
3. Prepare and start Control Service
4. Prepare and start Evaluation tool
5. Prepare and start Lock simulator
6. Connect locks from lock simulator
7. Begin transmission of messages from lock simulator
8. [Pause and close the Control Service]
9. [Restart the Control Service]
Between each step a delay is inserted to allow a steady-state to be reached in the Control Service, which simplifies comparison.

The test is executed for all indicated loads, as well as with the original Control Service, before the replaceable IPC layer was added. The Pause and restart is only performed for the IPC based test. When the IPC-based SBC Connection Repository is enabled it is done with the following parameter table:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SbcConnectionRepository</td>
<td>TestConnReposIPC</td>
<td>The IPC name of the SBCConnectionRepository. If omitted or empty it uses the default TCP based version.</td>
</tr>
</tbody>
</table>

All setup-files may be found in Appendix D – Test and configuration data.

**Results**

The results of the test can be summarized in Table 7. The table show the Availability and the Performance parameters when using the original Control Service (Original CS), the Control Service using the new design, but without enabling the SBC Connection Repository (No IPC CS), with the SBC Connection Repository enabled (IPC CS) and with the and the SBC Connection Repository after a restart of the Control Service (IPC Restart CS). All values are the average of the tests for the different loads.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Availability [ms]</th>
<th>CPU [%]</th>
<th>Memory [MB]</th>
<th>Threads [Count]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original CS</td>
<td>72,63</td>
<td>0,60</td>
<td>308,81</td>
<td>16,42</td>
</tr>
<tr>
<td>No IPC CS</td>
<td>70,16</td>
<td>0,41</td>
<td>309,15</td>
<td>15,42</td>
</tr>
<tr>
<td>IPC CS</td>
<td>69,85</td>
<td>0,46</td>
<td>325,79</td>
<td>25,83</td>
</tr>
<tr>
<td>IPC CS after restart</td>
<td>71,28</td>
<td>0,43</td>
<td>329,71</td>
<td>27,00</td>
</tr>
</tbody>
</table>

Table 7 - Scenario 2 results

For IPC CS and IPC Restart CS there are two processes, and therefore the CPU, memory usage and Thread count is the sum of the CPU, memory usage and thread count for each process. As the Virtual memory include a lot of shared memory between the two processes, the sum for memory is the sum of the Virtual memory of the Control Service and the private memory of the SBC Connection Repository service.

In Table 8 may be seen the difference in the results. Negative values indicate that the test shows that the performance or availability improved.

<table>
<thead>
<tr>
<th>Difference in %</th>
<th>Availability</th>
<th>CPU</th>
<th>Memory</th>
<th>Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original CS to No IPC CS</td>
<td>-3,38</td>
<td>-42,23</td>
<td>0,11</td>
<td>-6,08</td>
</tr>
<tr>
<td>No IPC CS to IPC CS</td>
<td>-0,46</td>
<td>65,66</td>
<td>5,38</td>
<td>67,78</td>
</tr>
<tr>
<td>IPC CS to IPC CS after restart</td>
<td>0,04</td>
<td>-5,76</td>
<td>1,21</td>
<td>4,51</td>
</tr>
</tbody>
</table>

Table 8 - Scenario 2 results difference

For the Original CS to No IPC there is quite a deviation for the CPU load. When looking at the actual numbers, this is primarily due to the very small value of the numbers. If the CPU load is offset by 10, so it becomes 10.6 and 10.41, then the deviation becomes a difference of 1,88%. For this reason, the CPU comparison will be ignored when the absolute values are so low. The other values are well within what could be accounted for by simple measuring fluctuations. It can therefore be concluded that the design of the dependency injected IConnectionListener interface only has a negligible affect on the Control Service, as was also expected.
When comparing the No IPC to IPC difference it is possible to see another large variation in CPU usage, which is ignored for the same reasons as the previous comparison – too small values.

The Memory and thread comparison show a large difference which is not due to small values. This is instead due to the multiple processes. When the SBC Connection Repository was introduced, a new process was spawned. Each process has a number of threads which all .NET processes utilizing thread pools have, regardless of whether they use them or not. Since both processes utilize thread pools, they both have the same minimum number of threads. This number can be modified, but that is considered outside the scope of this report. Instead, if we look at the absolute values they are approximately 10 higher than without IPC CS. This is due to the thread pool minimum thread count being 10. If we subtract 10 from the value we get a difference of 2.66% from No IPC to IPC. Naturally subtracting 10 is incorrect as some of these threads are used in the SBC Connection Repository, so we cannot just say the thread pool in SBC Connection Repository uses no threads. However, looking at the change in thread count as the number of messages increase (Figure 21), indicates that the thread pools are not overloaded, as there does not seem to be allocated extra threads to handle the extra messages. This indicates that the total needed thread count is less than the allocated. It may therefore be possible to tweak the thread pools of the two processes to obtain a thread count as before the separation into two processes, without hurting availability, but this is left for future work.

![Threads](image)

**Figure 21 - Scenario 2 Thread count comparison**

Some of the difference in memory consumption also come from the process allocation. Static variables and Framework objects are allocated in both processes. However, this is much more difficult to remove, as this memory is needed in both processes, and the performance penalty of 5.38% increased memory use must be considered an actual penalty.

The Control Service after restart consumes a little more resources than before the restart. It is believed this is due to latency in the Control Service clean-up, i.e. the garbage collection and thread pools, yet this has not been verified. Experiments with multiple restarts has shown that the memory consumption do not increase further after multiple restarts. Comparison of memory consumption for the different tests indicate that the increase is independent of message throughput, as may be seen in Figure 22.
Finally, there is Lock connections and messages. As the Lock Simulator is configured to not automatically reconnect if a connection is lost, it is clear from the test that the connections were maintained during the Control Service “update”. Furthermore, by comparing the total number of messages sent and the number of messages written to the database, it is possible to see that no messages were lost. The messages sent contain the transmission time set by the lock simulator, and, when the message is received by the Control Service, the reception time (LogTime) is tagged to the message. This is written to the database, and by querying the database it is possible to see that there is a gap in the LogTime while the Control Service was turned off, and then a burst of messages with same LogTime when the Control Service is started again. The transmission time stamp in the message indicates that the messages are transmitted in the same pattern during the Control Service being off and is received in order (can be seen from the Control Service message sequence number) when the Control Service is turned on again.

All the results may be seen in Appendix D – Test and configuration data.

5.2.4 Evaluation

The SBC Connection Repository is able to handle registering and unregistering the Control Service, and caching the messages received as well as any new or lost connections and is therefore able to allow the Control Service to be updated without the Lock connections being lost and having to be restored.

The availability of the IPC based SBC Connection Repository is almost identical to the version using direct TCP communication, indicating that the QoS requirement is met.
The performance parameters indicate that due to the extra process and the threads and memory needed by this extra process, it is not possible to meet the hypothesis requirement for a maximum 5% increase in performance parameters.

For this reason, the hypothesis is not fully met.

Further research should be done into using e.g. the RabbitMQ service as a backend. This has the advantage of relying on a tried and proven implementation, which limits the need for custom implementation and the risk of introducing mistakes or performance risks. It also has a much more complete implementation of registration, greatly simplifying the task of further dividing the Control Service into smaller more specialized services.

6 Conclusion

This report has focused on modifying an existing architecture, specifically with respect to availability and modifiability.

Specific scenarios have been formulated focusing on external dependencies (scenario1) and updating the application (scenario2), to prove or disprove the hypothesis.

During the research and evaluation of the REST-based external dependencies a lot of insight into the inner workings of the HTTP based communication protocol was gained, including the importance of protecting the applications against external dependencies, also known as SLA inversion.

The implementation of the SLA Circuit Breaker showed how it is possible to protect against SLA inversion without any other dependencies than the one causing the problem suffering, and it was done well inside the limits of the hypothesis.

It was also found that determining the optimal configuration parameters for the external communication and the SLA Circuit Breaker is very dependent on the domain in which the service exists. Choosing these parameters to be optimal will be left for future research, and to keep them optimal they may even need to be dynamically updated depending on behaviour of the external dependencies. However, for the scenario investigated in this report the chosen parameters were sufficient.

When it comes to updating the application the research quickly indicated that runtime updating already loaded types was not a viable solution, due to the level of complexity and risk of failure.

Instead, splitting the application into updatable parts was researched and found to be a usable solution. This report focused on the communication with the Locks, and here a division with a custom service holding the connections and communicating directly with the Control Service via IPC was chosen.

It was found that separating the Control Service into independent processes (services) can be done without hurting availability outside the 5% that was postulated in the hypothesis. It was also found to be quite possible to update the Control Service without losing any connections or messages.

During the research and evaluation of the options for separating the Control Service into updatable parts it was discovered that even though the custom IPC based separation functioned, a solution based on a COTS message queue may be beneficial. This is due to a number of functionalities needed by the communication backend for a large-scale implementation, and here the feature rich and tried and true nature of e.g. the RabbitMQ service would be an advantage, especially with respect to maintainability and testability.
As the basis for this report was an existing Control Service the report has taken as a premise that the modifications to the Control Service should be localized as much as possible to improve testability and to minimize the risk of introducing errors during the modification of the Control Service. It was also decided that the modifications should be configurable, meaning they can be disabled or enabled in the Control Service. These decisions have helped to focus on clear and concise interfaces and classes with well-defined responsibilities.

After the writing of this report an experiment on actual production data was performed. The results do not alter the conclusion or evaluations, but in the interest of completeness the obfuscated real data and test results has been added to the test data of Appendix D – Test and configuration data.

7 References

[1]: Unknown origin. Many have been sited with this quote or a variation of it, but none can claim to be its author.
[3]: Michael Nygard, Stability Patterns ...and Antipatterns, second edition, 2012
[8]: http://www.mbtest.org
8 Appendix A – Realistic message activity
This appendix shows the queries made to determine the approximate values used for the Deployment data. This was executed on 5/2-2018 at 17:50.

Removed due to confidentiality

9 Appendix B – Communication choices
This Appendix includes an overview of the current communication choice of the key parts of the current Software Architecture as well as their approximate load.

9.1 Internal communication channels
Currently the communication internally is piped IPC, .NET remoting and MySQL Connector/Net

9.2 External communication channels
- SMS Receiver calls a built in REST server
- Emergency Call uses TCP.
- Smartphone App uses TCP.
- Lock uses TCP/UDP
- Supported devices uses TCP/UDP
- SMS Transmitter use http
- Email transmitter uses SMTP
- Client REST server uses http with TLS and basic authentication (optional)
- Client application uses http with TLS and basic authentication
- Client browser uses HTML
- Email receiver integrates with booking and registration systems to generate users via email, as well as
- NemID and CPR service allows login with NemID, as well as extraction of address information about the person logging in.
- AD FS service allows for login via AD FS so the customer can use their own AD for validation.
- SIP service handles ingoing calls from emergency call system (DTMF based) and is used for notification in case of an error. May also be used for support call handling.
- Service monitor looks into all running processes and monitors threads, CPU, RAM, BIT status as well as overall server parameters. It can shut down and restart services automatically.
- Script runner is an assortment of scripts created by request of clients to have custom functionality run periodically, e.g. generate reports, monitor suspicious behaviours, synchronize with client database and create/remove users based on this. Monitor for client rest events or periodically poll client REST API to create/remove users.

Some details in above removed due to confidentiality

10 Appendix C – Tool description and configuration
This appendix describes the testing tools created to evaluate the hypothesis.
10.1 Lock simulator

The Lock simulator (Figure 23), also known as the SBC Simulator, hence the programming form title, is used to simulate locks to the Control Service. The Lock simulator was an existing tool, but has been augmented with an Event transmission runner, allowing for transmitting many events in a short period of time. This may be seen in Figure 24.

The tool uses a configuration file to setup some of the default values. Yet all these values can also be entered in the tool.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DefaultHost</td>
<td>The host address of the Control Service</td>
</tr>
<tr>
<td>DefaultPort</td>
<td>The port where the Control Service is listening.</td>
</tr>
<tr>
<td>DefaultUnit</td>
<td>The path to the file containing the locks to simulate, one lock id on each line.</td>
</tr>
<tr>
<td>DefaultEventFile</td>
<td>The file containing the events to simulate from the locks</td>
</tr>
<tr>
<td>DefaultLogFile</td>
<td>The .csv file to write logging to.</td>
</tr>
</tbody>
</table>

10.2 Performance and Availability evaluation tool (PAAET)

This tool is a simple console application. The application uses a configuration file to setup the values needed to perform the test. The configuration files parameters are shown below.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ControlServiceConnectionString</td>
<td>The connection string for the database. This is used for checking if the event notification is already registered or must be registered</td>
</tr>
<tr>
<td>controlservicehost</td>
<td>The host address of the Control Service used to connect the Lock simulator needed to measure availability. This should always be localhost</td>
</tr>
<tr>
<td>controlserviceport</td>
<td>The host port of the Control Service used to connect the Lock simulator needed to measure availability.</td>
</tr>
<tr>
<td>controlserviceunit</td>
<td>The id of the unit used for availability test. Default: paaetmonitorunit</td>
</tr>
<tr>
<td>controlserviceipc</td>
<td>The IPC name used to communicate with the Control Service to update access code for availability test. Default: TestControlServiceIPC</td>
</tr>
<tr>
<td>controlservicecustomer</td>
<td>The customer owning the Control Service unit used for availability test: Default: paaet_testcus</td>
</tr>
<tr>
<td>controlservicepath</td>
<td>The path to the Control Service executable or Control Service and SBC connection repository executable</td>
</tr>
</tbody>
</table>
separated by semi-colon. This is used for Performance value measurement.

<table>
<thead>
<tr>
<th>LogFile</th>
<th>The path to the .csv file where the results should be written.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SampleInterval</td>
<td>The interval in ms between each sample of performance parameters. Default: 1000ms</td>
</tr>
<tr>
<td>SampleTime</td>
<td>The time in seconds to sample the performance parameters for each execution of the sample command line command. Default: 30s</td>
</tr>
<tr>
<td>AvailabilitySamples</td>
<td>The number of availability measurements to perform for each sample command. Done both before and after the performance parameters sample. Default: 5</td>
</tr>
</tbody>
</table>

The tool has three commands, as described below, but is also described in the tool, as can be seen in Figure 25. The commands are not case sensitive

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAMPLE</td>
<td>Perform a sample, which consists of an availability test, followed by a performance test and then an availability test again.</td>
</tr>
<tr>
<td>PEXIT</td>
<td>Only used for scenario 2 test. Execute a Pause listener on the Control Service, which in turn informs the SBC Connection Repository that it should expect the Control Service to shut down and start caching events until it comes back.</td>
</tr>
<tr>
<td>EXIT</td>
<td>Close the console application</td>
</tr>
</tbody>
</table>

![Figure 25 - PAAET execution example](image)

11 Appendix D – Test and configuration data

This appendix contains the digital data enclosed with the project. The source code, the test result data, etc. Please refer to the following DropBox link to download the .zip file. The .zip file is password protected with the password: Maa2018
The tests each have a setup file which sets up the test for local execution. Naturally to reproduce on multiple servers the information in the setup file must be altered to not use localhost but instead use the actual host name. The host name of the rest client simulator is indicated in the beginning of the setup file:

- `restHost` Indicates the server address of the REST client simulator.
- `csHost` Indicates the server address of the Control Service.
- `csPort` Indicates the port on the server to use for the Control Service.

The setup files generate the database and copies the required tools and the controlService to the execution directory. It also prepares the configurations files for the tools and Control Service to use the desired configuration files and sets the log output directories inside the execution folder. Each setup file generates its own directory inside the execution directory. Experiments with the setup files has only been done where the directory path contains no spaces or special char.

The digital part of the report has the following directory structure:

- `controlService` The Control Service executable and dependent libraries.
- `execution` This directory is generated by the setup files and will contain the test executions. The content is taken from the other directories.
  - Scenario<N> Contains all files required to execute the test including batch files to start the test and modify the tools.
    - `controlService` Contains a copy of the Control Service prepared for the selected test
    - `results` This is where the results of the test are placed
    - `setup` Contains the files for configuring the lock simulator
    - `tools` Contains a copy of the needed tools prepared for the selected test
  - `results` This directory contains all the results from the tests entered in excel sheets. <N> is either 1 or 2, indicating the scenario. <Z> is either 2 or 5000 depending on the number of locks. <P> is ether 3, 15 or 75 because the tests for scenario 1 was entered into different documents, where for scenario2 they are entered in the same document. Scenario<N>._combined contains the combination of data not found in the other documents used for the report. Finally the scenario<N>._real contains the data for the test executions on real data.
    - `scenario1_graphs_<Z>locks_<P>.msg.xlsx`
    - `scenario2_graphs_<Z>locks.xlsx`
    - `scenario<N>._combined.xlsx`
    - `scenario<N>._real.xlsx`
  - `scenario1`
Contains the setup data for scenario 1
  - prepare_execution.bat
    Prepares the execution folder and database for scenario 1 test with 2 locks.
  - prepare_execution_5000.bat
    Prepares the execution folder and database for scenario 1 test with 2 locks.
  - Go to execution
    Link to scenario 1 execution folder
  - setup
    Contains setup files needed for the scenario 1 test.
    - db_setup.sql
      Contains the database initiation for scenario 1 with 2 locks
    - db_setup_5000.sql
      Contains the database initiation for scenario 1 with 5000 locks
    - restclient<n>.bat
      These are copied to the execution folder. Each batch file commands the mountebank tool to create or update the three listeners with one of them having a delay of <n> seconds.
    - run_1_start.bat
      Copied to the execution folder. Used to initiate the first test (no SLA Circuit Breaker and 300s REST timeout).
    - run_2_slac.bat
      Copied to the execution folder. Used to initiate the second test (SLA Circuit Breaker and 300s REST timeout).
    - run_3_25s_timeout.bat
      Copied to the execution folder. Used to initiate the third test (no SLA Circuit Breaker and 25s REST timeout).
    - run_4_slac_25s_timeout.bat
      Copied to the execution folder. Used to initiate the fourth test (SLA Circuit Breaker and 25s REST timeout).

- scenario2
  - prepare_execution_5000.bat
    Prepares the execution folder and database for scenario 2 test with 5000 locks.
  - Go to execution
    Link to scenario 2 execution folder
  - setup
    Contains setup files needed for the scenario 2 test.
    - db_setup_5000.sql
      Contains the database initiation for scenario 2 with 5000 locks
    - run_1_start.bat
      Copied to the execution folder. Used to initiate the first test (No IPC Lock Connection).
    - run_2_ipc.bat
      Copied to the execution folder. Used to initiate the second test (IPC Lock Connection).
    - run_3_ipc_restart.bat
      Copied to the execution folder. Used to start the Control Service and PAAET tool after PEXIT in PAAET and the Control Service was closed (IPC Lock Connection continued).

- setup_shared
  - createRawDb.sql
    SQL query to generate the schema of the database
• lock_sim_events_<N>.xml
  Configuration file for the lock simulator to simulate messages sent from locks. This is used in all simulated tests. <N> indicates how many events are simulated a second.
• lock_sim_ids.txt
  Configuration file for the lock simulator to simulate a collection of locks, in this case 2 locks
• lock_sim_ids_5000.txt
  Configuration file for the lock simulator to simulate a collection of locks, in this case 5000 locks
• tool
  o LockSimulator
    The Lock simulator including a generic configuration file which is modified during
  o PerformanceAndAvailabilityEvaluationTool
    The PAAET or Evaluation tool used to evaluate the availability and performance of the tests.
  o ReplaceText
    A vb script used by the setup files to configure the configuration files by text replacement. Used for test setup, not part of the actual test
• source code
  o PerformanceAndAvailabilityUnitTest
    Contains unit tests relevant to the scenarios.
  o PerformanceAndAvailabilityEvaluationTool
    The PAAET or Evaluation tool used to evaluate the availability and performance of the tests.
  o SBCProtocolLibrary
    Library containing the code for Scenario 2
  o NotificationLibrary
    Library containing the code for Scenario 1
• unittest
  Unittest binaries
• real
  This directory contains the same setup as for scenario 1 and scenario 2, except the database and events are generated from the actual production database and communication log files. The data is obfuscated to remove any confidential information. As the server with a high number of REST receivers (used for scenario 1) has a much lower number of Locks, the server with the highest number of locks is used as basis for the test of scenario 2 on real data. The test data in the report is a combination of message count from the server with maximum message throughput and the lock count from the server with the highest lock count.

12 Appendix E – Race condition test
This appendix shows the results of the tests performed on the original Control Service before the race conditions was discovered and fixed. This is only included as information of interest and to indicate why the single Control Service testing was chosen, and refer only to scenario 1

Modifying the REST service delay is done as follows. In parenthesis may be seen the time at which the delay was introduced/ altered.

• Introduce 10s delay on noification1 listener (191.547,1ms into the test)
• Introduce 10s delay on noification2 listener (243.547,2ms into the test)
• Introduce 10s delay on noification3 listener (278.547,3ms into the test)
• Introduce 10s delay on noification4 listener (310.547,4ms into the test)
• Introduce 10s delay on noification5 listener (334.547,4ms into the test)
• Alter notification1 listener delay to 30s (362.547,5ms into the test)
• Alter notification2-5 listener delay to 30s (369.547,5ms into the test)
• Introduce 30s delay on notification6 listener (419.547,6ms into the test)
• Introduce 30s delay on notification7-10 listener (439.547,6ms into the test)

The CPU data shows the number of ms the Control Service occupied the CPU between samples. As the sample frequency is 1s it can be approximated that if 800ms of CPU time is used since the last sample, then the Control Service has occupied the CPU 80%. The Red lines are the actual measurements and the blue is a 10-sample running average. This may be seen in Figure 26.

The memory consumption in byte can be seen in Figure 27 and here it is very obvious how the accumulation of incomplete REST events are consuming more and more memory.
As the notification to the REST client is performed asynchronous in the operating system network layer seen from the Control Service, the number of worker threads needed by the Control Service is not greatly affected by the delay, until the Control Service experience a synchronization dead-lock, and therefore allocates more and more threads, as may be seen in Figure 28. In Figure 29 may be seen the thread usage before the catastrophic failure, giving a better idea of the Control Service thread usage during REST client service delay, where it may be seen that it is not significantly affected by the delay.
The availability test after the deadlock failure shows no response after 60 seconds – most likely an infinite response time. The response time during the delay period to both the REST event and code change is relatively unaffected, and the availability do not suffer significantly until the deadlock failure.

13 Appendix F – History of external functionality

Separating specific functionality and moving functionality into their own processes is not new, and a little history about previous and current approaches will be mentioned here, from the current micro-service architecture over Software as a Service (SaaS) to SOAP.

At the time of this report micro-service architecture is the latest in a long line of attempts to achieve low coupling between software components. In Micro service architecture the idea is to build the software component as a collection of small independent services, which can then be deployed and updated as needed.

SaaS is where some software functionality (library) is made available (for a price) to users. SaaS differs from Platform as a Service (PaaS) and Infrastructure as a Service (IaaS) by having a very clear and defined functionality (and hopefully responsibility).

Before that the original idea of Simple Object Access Protocol (SOAP) was not only to have a defined way of performing Remote Procedure Calls (RPC), but via Universal Description, Discovery and Integration (UDDI) and Web Services Description Language (WSDL), it was envisioned that users (or developers) could discover the
desired functionality (library) at runtime and use it, without caring about where in the world the actual computation was done or how it was achieved.

14 Appendix G – IPC single image diagram

![Figure 30 - IPC based lock connection sequence diagram]

15 Appendix H – UnitTest and results

The Unittests may be seen in the source code in Appendix D – Test and configuration data, and the results of an execution are shown in Figure 31. The unit tests are named according to what they test, and the individual tests will therefore not be described further.
15.1 SlaCircuitBreaker
The SLACircuitBreaker relies on time to determine when a pending execution is delayed, and when it is time to trip the relay and when to try and reset it. For this reason, a test stub for IDateTimeFactory is created and used instead of the default DateTimeFactory. When this test stub is dependency injection into the circuit breaker it allows the test to simply set how much time has passed instead of relying on sleeps.
15.2 SBCConnectionRepository

The IPC separation of the Lock connections and the Control Service uses not only the TCP implementation, the IPC implementation, but also threading. These are stubbed by implementing a stub for IConnectionListener returning stubbed connections instead of real ones. The connections themselves use the ISBCProtocolStream, which is also stubbed thereby completely removing the dependency on TCP. The IPC layer is also removed by using special testability methods on the implementation which simulated IPC within the same process. Finally, the use of the thread pool is also removed by enabling testing on the thread pool, which simply executes everything in the calling thread.

15.3 DBCachedExecution

This was removed from the project and is only included to avoid removing it from the project for this report. Please disregard these tests.

16 Appendix I –REST communication considerations

In order to use windows as a REST client and a REST server (not same machine, but both windows) there are some important aspects built into the underlying web service implementation (Internet Information Service (IIS) and Windows Communication Foundation (WCF)) as well as the REST client (RestSharp and HttpWebRequest).

The REST client implementation has a built-in protection preventing too many simultaneous ongoing asynchronous calls to a given host. If too many simultaneous calls are started the excess calls are simply queued. This mechanism makes sense to have. If many simultaneous requests to the same server is started, and the server is already loaded by other requests, it will be slow to respond and initiating even more simultaneous requests may cause it to get overloaded. Furthermore, each ongoing request must reserve a TCP port on each side. There are theoretically 65536 numbers, but several are reserved; well-known = 0-1024, registered = 1025-49151, leaving the last 16383 ports (49152-65535), called dynamic ports, to use for open connections. If a process is allowed to create unlimited simultaneous requests, the ports could quickly be exhausted. Under normal circumstances built-in protection is not an issue, because the receivers are distributed over many servers, but during testing it is needed to redirect many calls to a single host. There is a configuration parameter to alter the permitted number of simultaneous calls (System.Net.ServicePointManager.DefaultConnectionLimit) and this one is set on the Control Service during testing to 1000 simultaneous calls (configuration parameter RESTSameHostNotificationLimit).

On the IIS/WCF side similar protection exist ensuring that not too many simultaneous ongoing calls can exist on a given web service. The same reasons regarding stability with many simultaneous calls and port limitation as above apply. Furthermore, many simultaneous calls being executed in different threads may spend a large part of the available processing power on synchronization and thread switching.

The protection may be configured in several ways affecting the way the incoming requests are processed. One is the ConcurrencyMode. This determines how incoming requests from the same client is handled, and the default is Single, meaning only a single request from a given client will be permitted at a time. Any requests from a client which already has a request being processed will be queued and processed in order. This may be altered by changing the ConcurrencyMode to Multiple. Another configuration is the InstanceContextMode, which specifies when to create an instance of the service object. By choosing PerCall it triggers a new instance of the service object for each call, yet as long as the implementation can handle multiple simultaneous calls to the same object method, it can remain Single. Unfortunately, this is not enough. Calls are processed by a ThreadPool which exists inside the executing process (often IIS). This means that once the allocated threads are all in use the remaining
requests will be queued. The ThreadPool will schedule new threads (default one per second) until some configured maximum, but it means that if the REST receivers use the same process to host the service objects, then once one service object slows down, it will affect all other service objects hosted in the same process. To avoid this, it is necessary to deploy the individual endpoints in their own process.

In Mountebank, WCF is not used as backend, and therefore the receiving side limitation do not exist. However, the principles are still the same. All receivers on a single port affect each-other, and it is therefore necessary to use a different port for each receiver, and then they will be independent.

Finally, there are some important aspects about the REST notification implementation which should be considered. The implementation is, as mentioned earlier, based on the RESTSharp, which in turn is based on HttpWebRequest. This class operates with two independent underlying timeouts; Timeout and ReadWriteTimeout.

The first value Timeout indicates how long the request will wait for the connection to be established and the second value ReadWriteTimeout indicates how long the request will wait for the response to complete after sending its request. The default value for Timeout is 100 seconds and the default value for ReadWriteTimeout is 5 minutes (300 seconds). These values are unchangeable in the original implementation due to limitations in the interface yet has been made configurable in a later release.

All testing will be performed with both the original timeouts to keep the behaviour of the original implementation, as well as a much smaller timeout. Naturally tweaking these values more may have a positive effect, yet this will be considered outside the scope of the report to evaluate.

### 17 Appendix J – Selected graphs

This appendix contains graphs, which are considered to have relevant information, yet only as a background to the tables and graphs in the main report. For each graph will be described what observations of interest it shows.

For a complete view of all graphs and data please refer to Appendix D – Test and configuration data.
17.1 Scenario 1 CPU with no delay

Here it may be observed how the CPU usage is almost completely independent on the number of locks connected but has an almost linear dependency on the number of message. Each time the number of message increase by a factor of 5 the CPU usage increase with almost the same factor.

17.2 Scenario 1 Memory as REST received delay increase, 15msg/s

Here it may be seen how the memory consumption begin to increase greatly when the REST transmissions begin to fail, and that this increase in memory persists even after the delay has been removed. This same pattern can be observed on the other graphs with fewer or more messages a second.