AARHUS UNIVERSITET

SOLIDUM PETIT IN PROFUNDIS · UNIVERSITAS ARHUSIENSIS

DATALOGISK INSTITUT

DET NATURVIDENSKABELIGE FAKULTET

Afgangsprojekt

Master i Informationsteknologi

Linien i Softwarekonstruktion

Different Paths to High Availability by Introducing Redundancy in a Distributed SCADA System

af

Morten Andersen
Årskort 19941458
14.juni 2011

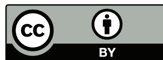
Morten Andersen, studerende

Henrik Bærbak Christensen, vejleder

Copyright Notice

©2011 Morten Andersen

This master thesis is non-commercial academic work. It would only be a pleasure to the author if you want to use or build upon any part of the ideas, analyses or prototypes presented here, whether for academic, commmercial or non-commercial work. So, *use as you see fit*, or in legalese:



This work is licensed under the Creative Commons Attribution 3.0 Unported License. To view a copy of this license, visit http://creativecommons.org/licenses/by/3.0/

Different Paths to High Availability by Introducing Redundancy in a Distributed SCADA System

Morten Andersen, Årskort 19941458
Master Thesis
Master of Information Technology, Software Development
Department of Computer Science, University of Aarhus
8200 Århus N, Denmark

June 14, 2011

Abstract

This thesis documents the architectural theory in adding new high availability requirements to an existing highly distributed Java based wind farm Supervisory Control and Data Acquisition (SCADA) system. A soft real-time system with an architecture that was originally designed with performance as the main architectural driver.

Based on the theory three different prototypes have been build, each using a different path to reach the same goal. One using an active redundancy tactic with network multicast, one using passive redundancy using distributed shared memory (DSM) and finally one prototype build upon the end-to-end principle.

To test how the three prototypes performs when applied to the SCADA system, a test bench using virtual machines has been build running on the Amazon Elastic Compute Cloud (EC2) computing platform. This test bench made it possible to evaluate the three prototypes in a realistic way running on a distributed system with 18 nodes without a big hardware setup.

The tests revealed that the DSM based prototype failed in fulfilling the new requirements. The prototype based on network multicasts was almost successful, with some minor bugs that will need additional work. Finally the end-to-end based prototype was very successful in adding the new availability requirements.

Contents

1 Motivation **7**
1.1 The Distributed Wind Farm 8
1.2 The Aim of the Thesis 10
1.3 Problem Statement 11
1.4 Delimitations 11
1.5 Overview of the Thesis 12

2 The Distributed Wind Farm **12**
2.1 Functional Requirements 12
2.2 Architectural Description of the Existing System 13
2.2.1 Component and Connector View 14
2.2.2 Allocation View 17
2.2.3 Module View 17
2.3 The System Components in Client-Server Terms 19
2.4 Distributed Data Structures 21
2.5 Architectural System Quality Attributes 21
2.5.1 Architectural Drivers 22
2.5.2 Quality Attribute Scenarios (QAS) 22
2.6 Source Code for the System 25

3 Availability Theory **26**
3.1 Availability - The Uptime Percentage 26
3.2 The End-to-End Principle 26
3.3 Failure Semantics Theory 27
3.3.1 Choosing the Relevant Faults 28
3.3.2 Handling Faults - Mask, Propagate or Correct 28
3.3.3 Cooperative Masking: Hierarchical 29
3.3.4 Cooperative Masking: Group 29
3.4 Tactics for Increasing Availability 31
3.4.1 Voting 31
3.4.2 Spare 32
3.4.3 Active Redundancy 32
3.4.4 Passive Redundancy 32
3.5 Replication Theory and Tradeoffs 32
3.5.1 Types of Replication Messages 34
3.5.2 Group Communication - Multicast Messages 35
3.5.3 Memory Consistency Models 36

4 Applying the Theory to the Wind Farm SCADA System **40**
4.1 Applying the End-to-End Principle 40
4.1.1 End-to-End Notification 40
4.1.2 End-to-End Polling 41
4.2 Analysis of Applying the Different Availability Tactics 42
4.2.1 Failure Semantics 42
4.2.2 Availability Tactics 42
4.2.3 An Active Redundancy Tactic 43
4.2.4 A Mix of Active and Passive Tactics 45
4.2.5 A Passive Redundancy Tactic 48

- 4.3 Applying the Replication Theory . . . 50
  - 4.3.1 The Types of Replication Messages . . . 50
  - 4.3.2 Failure Semantics for the Group Communication . . . 50
  - 4.3.3 Memory Consistency Models . . . 50

5 Solution for the External Group 51
- 5.1 Failure Detection . . . 51
- 5.2 Recovery . . . 51

6 Solutions for the Internal Group 52
- 6.1 Transparency of the Solutions . . . 53
- 6.2 A Passive Redundancy DSM-Based Solution Using Terracotta . . 55
  - 6.2.1 Implementation Notes . . . 57
  - 6.2.2 Analysis of Pros and Cons . . . 58
- 6.3 An Explicit Active Redundancy Solution Using Hazelcast . . . 59
  - 6.3.1 Implementation Notes . . . 60
  - 6.3.2 Analysis of Pros and Cons . . . 61
- 6.4 An End-to-End Solution with Soft State . . . 62
  - 6.4.1 Conceptual Description of the Solution . . . 62
  - 6.4.2 Implementation Notes . . . 63
  - 6.4.3 Analysis of Pros and Cons . . . 64
- 6.5 Solutions Not Evaluated . . . 65

7 Test Bench and Test Method 65
- 7.1 Physical Setup . . . 65
  - 7.1.1 Number of Nodes . . . 66
  - 7.1.2 Practical Problems in Managing the Nodes . . . 66
  - 7.1.3 Potential Problems in a Virtualized Solution . . . 66
- 7.2 Test Receipt . . . 69
  - 7.2.1 Algorithm for Random Crashes . . . 70
- 7.3 Measurement Method . . . 72
  - 7.3.1 Data to Collect from Clients . . . 72
  - 7.3.2 Analyzing Data . . . 72
- 7.4 Alternative Measurement Method - Qualitative Network Packet Analysis . . . 73

8 Evaluating the Prototypes 73
- 8.1 The Terracotta Based Prototype . . . 75
- 8.2 The Hazelcast Based Prototype . . . 76
- 8.3 The End-to-End Based Prototype . . . 79

9 Conclusion 81

Appendices

A Table of Contents of the Source Archive 83
- A.1 Source Code for the Prototypes . . . 83
- A.2 Results of the Test Runs . . . 83
- A.3 Other Artifacts . . . 83

B Building and Running the Software **84**
B.1 Prerequisites 84
B.2 Download 3. Party Libraries 84
B.3 Setup an Amazon EC2 Account 84
B.4 Create an Amazon EC2 Ubuntu 10.04 Image 85
B.5 Building Using Gradle 85
B.6 Eclipse Launch Files 85

C Manual for Reading the Source Code **86**
C.1 Wind Turbine Service Source Code 86
C.2 Wind Farm Service Source Code 87

D Java Program for Automating the Test Runs **87**
D.1 Management Console Commands 88
D.2 Controller Application Webservice Operations 89

E Scripts for Automating the Test Bench Setup **89**

F Qualitative Analysis - RMI Packet Analysis **90**
F.1 Hypothesis on Packet Numbers and Sizes 91
F.2 Measured Packet Numbers and Sizes 91

G Test Report for the Hazelcast Based Prototype **95**

H Test Report for the End-to-End Based Prototype **95**

References **97**

1 Motivation

An often used architectural style in distributed systems is some variant over the classical client-server architecture, where a single server serves a multitude of clients. A stylistic example of such an architecture is shown in figure 1. As it is seen all communication paths in the system goes through the single server node, making this a critical single point of failure.

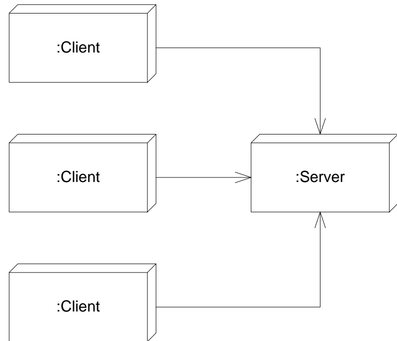


Figure 1: The client-server architecture with a single server.

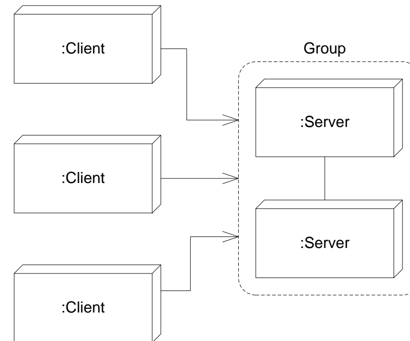


Figure 2: Redundancy in the client-server architecture.

For systems where the clients are unable to perform the required business logic of the system if the server node is unavailable, an architecture like this is of course extremely fragile, since a breakdown of the central node will have severe consequences on the availability of the entire system. An extreme example of such a system is the original client-server architectures using a central mainframe and terminals as clients, where all processing stops if the central mainframe is unavailable.

In systems where high availability is necessary it will therefore often be necessary to introduce some kind of redundancy of the central server node. A list of potential architectural availability tactics for so called fault recovery is described in [Bass et al., 2003, ch.5.2]. For distributed systems this spans from tactics with no downtime like voting and active redundancy, over short downtime tactics like passive redundancy to tactics with longer downtime like having a standby spare system. In this thesis the focus is on the online no downtime tactics.

For client-server systems with simple communication patterns, where the protocols used are *stateless*, and the role of the server node is simply to process and respond to individual idempotent client operations, it is relatively straightforward to introduce redundancy. This can simply be done by adding several server nodes, and using some kind of routing layer to make it transparent to the application logic in the clients, which one of the several server nodes are actually used. A stylistic view of a system with such a *group* of server nodes is depicted in figure 2.

Unfortunately the complexity of introducing several server nodes grows when the communication patterns and protocols has states and steps, and models some kind of longer relationship between the client and server nodes. Examples of such communication patterns are:

- Distributed transactions.
- Asynchronous remote method invocations - i.e. with callbacks.
- Subscription based protocols, e.g. a distributed version of the observer pattern ([Gamma et al., 1995]).

In the case of such *stateful* systems some kind of communication between the server nodes are necessary, which is shown as the communication link between the two example server nodes in figure 2.

In the academic world the theory of high availability systems is a well described problem, for which several solutions exists each with well defined theoretical limitations of what fault scenarios they will handle gracefully. In the software industry when one as a practitioner needs to select between and apply one of these high availability solutions to a system, the actual implementation must occasionally be shoehorned onto an existing system as an after the fact architectural requirement.

These after the fact situations typically occurs as a natural evolution of a system when it transitions from being "just" an important standalone system, to being a crucial infrastructure system that other systems depends on as being highly available. A change of the systems role meaning that the architectural requirements for the system suddenly will include high availability requirements.

The following section gives a high level overview of a real-world stateful client-server system where this evolution into a high availability system is required. The requirement of bolting on high availability of the central server node in this system, is exactly what was the motivation for this thesis.

1.1 The Distributed Wind Farm

The distributed wind farm system that acts as the case system for the theory in this thesis, is depicted in an informal rich picture notation in figure 3 on the following page. This picture shows the nodes in an anonymous *supervisory control and data acquisition* (SCADA) system for monitoring and management of wind turbines in a wind farm. This is a typical example of a distributed industrial control system where client access to sensors and actuators on several physical units (here wind turbines) takes place through a single central server, in this case labelled the *wind farm server*.

One of the functionalities in the actual system is that client applications can subscribe to receiving soft real-time values from one or more wind turbines, or in more formal sentences:

> To monitor soft real-time values for a client specified list of sensors in a turbine. The soft deadline for the periodic readings is one second.

To fulfill the performance aspects of this requirement it has been necessary for the system to avoid sending lots of unchanged sensor readings through the system from the turbines, over the central wind farm server to the clients. Wherefore the distributed communication has been implemented using a distributed version of the standard observer pattern in the *push* variant[1] using remote pro-

[1] See [Gamma et al., 1995] for detailed treatment of the observer pattern and the difference between the *push* and the *pull* model. Basically in the *push* model the notification-messages contains details about the actual changes (e.g. such as the new value of a sensor), whereas in the *pull* model the messages only informs the subscriber that a value has changed.

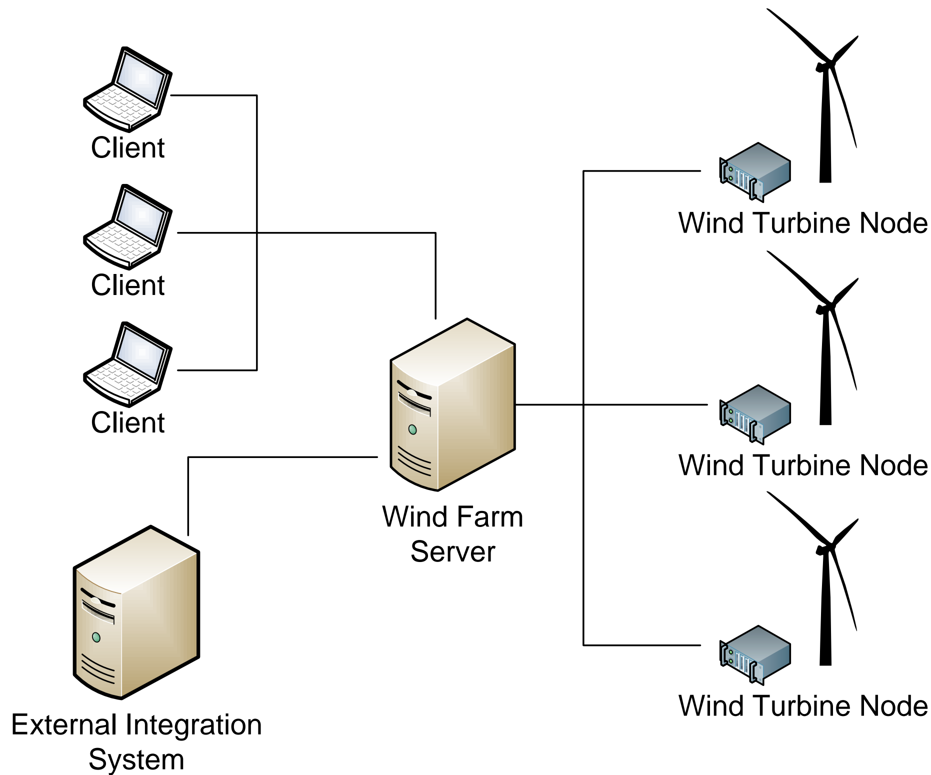


Figure 3: Informal overview of the wind farm SCADA system.

cedure calls (RPC).

To further minimize the network traffic in the system, the central wind farm server acts as a subscription multiplexer, combining several client system subscriptions for the same sensor value, into only one subscription on the target wind turbine node. Unfortunately this mechanism leads to quite some protocol housekeeping *state* being introduced in the central node, making it harder to introduce redundancy of the central node in a simple way as this *state* will have to be distributed to the redundant copies of the central node.

The central mechanism in this distributed observer pattern is depicted in figure 4 on the next page where the wind farm server acts as a stateful *forwarding observer*[2] of events from the turbine nodes to the end clients. For this introduction to the problem, the reader is urged to overlook the fact that the wind farm server in the figure does not actually forward events to the clients. Instead the client actively polls the wind farm server for changed values. This is due to some limitations in the technology used in the system, and will be explained in depth in later chapters when the system is analysed in depth.

Besides the central mechanism in the system, the figure also shows the problem in increasing the overall system availability by naively adding additional

[2]See [Coulouris et al., 2005, ch. 5.4.1] for a description of the nomenclature used in distributed event systems. A *forwarding observer* basically acts as a subscriber for events on behalf of the actual subscriber(s), and therefore forwards event notifications to the end subscriber(s).

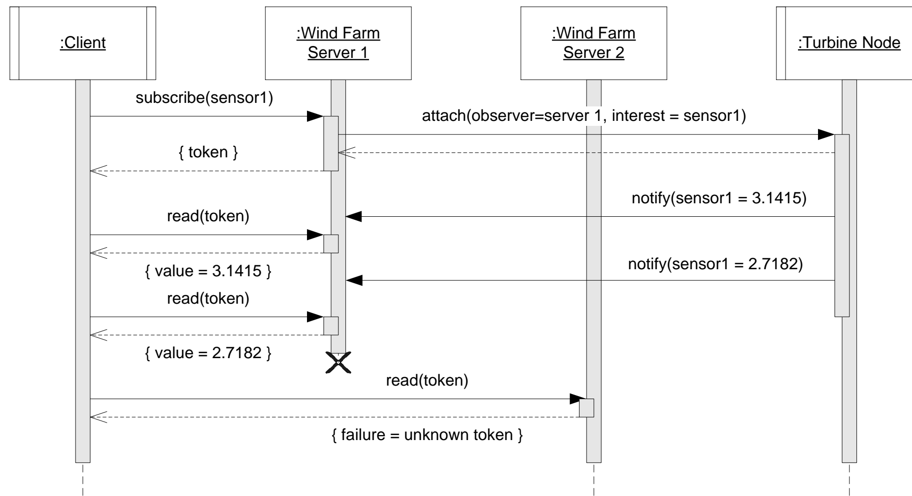


Figure 4: Stateful observer pattern.

wind farm server nodes (a redundant node labelled *wind farm server 2* is depicted in the figure). Due to the state present in the *wind farm server 1*, the second server is unable to handle the final client `read(token)` call depicted on the figure due to at least two obvious factors:

Unknown client token: The client token and the session it represents are unknown on the second server if no additional communication is added to the system.

No notifications to server 2: Since the second server node has not attached itself as an observer for the relevant sensors on the turbine node it naturally does not receive any notification messages.

1.2 The Aim of the Thesis

Based on the introduction of the wind farm SCADA system in the previous chapter, the aim of this thesis is to do a high level exploration of what kind of theoretical solutions are applicable for transforming the existing system into a highly available system where the central wind farm server node is no longer a single point of failure. This of course includes introducing redundancy of this central node, but increasing availability requires more than just redundant nodes, as best described in the quote by Bass et al.:

> *All approaches to maintaining availability involve some type of* ***redundancy****, some type of* ***health monitoring*** *to detect a failure, and some type of* ***recovery*** *when a failure is detected.*
>
> *[Bass et al., 2003, ch.5.2]*

Based on this theoretical exploration, it is the plan to apply several types of solutions to a working model of the distributed wind farm architecture, to analyze how suitable they are.

The work described in this thesis should therefore hopefully both act as a learning vehicle for the author, and for the reader, it should act as a catalog of ideas for how to attack the problem of introducing redundancy in systems with a similar distributed architecture and similar architectural qualities as described in chapter 2. Furthermore it is the aim to build a test bench that can be used for testing solutions for distributed systems with many nodes like the wind farm SCADA system in a virtual environment.

1.3 Problem Statement

The problem statement for the work described in this thesis is:

> To *analyse* how the theoretical high availability tactics can be applied to the distributed wind farm SCADA system as an evolution of the existing system. Based on this analysis, to *apply* several of these theories as architectural prototypes on a model of the distributed wind farm system. And finally, to *evaluate* the suitability of the applied solutions.

A couple of the terms in this compact statement might need a few additional words, for a concise definition of what they mean in the context of this thesis:

Distributed wind farm SCADA system: The specific system introduced in the motivation for this thesis. See chapter 1.1. A simplified model of this system, described in chapter 2, will be used in this work.

Evolution of existing system: As the system has already been running for a long period (approaching it's 10 years anniversary) it is an important factor in the analysis of the suitability of the different tactics, to consider how they can be applied to the existing system as an evolutionary change instead of a revolutionary change, i.e. as well as weighting the availability characteristics of the different solutions, the ease of how the solution can be "bolted on" to the existing system must be taking into account when a solution is evaluated.

Architectural prototype: A running prototype of limited parts of a system build to explore and evaluate a given architecture, or as described by [Bardram et al., 2004] *"... a learning and communication vehicle used to explore and experiment with alternative architectural styles, features, and patterns in order to balance different architectural qualities"*.

1.4 Delimitations

As the thesis work is based upon finding solutions for a specific system, there are certain natural assumptions and delimitations for what this work will cover:

Java based: The specific system under analysis is a Java based system. Therefore the implementation part of this thesis will be done using the Java platform.

No Transactions: The wind farm SCADA system is of a nature where distributed transactions are not relevant. So, although distributed transactions are a very relevant problem in certain systems when introducing redundancy, it will not be covered in this work.

1.5 Overview of the Thesis

The thesis starts out in chapter 2 with a detailed architectural description of the case system used in the thesis. The chapter ends with a formal description of the availability requirements that must be added to the system in chapter 2.5.

Thereafter the relevant theory on adding availability to a distributed system is described in chapter 3, followed by chapter 4 describing how the theory is applied to the actual wind farm SCADA system.

Chapter 5 and 6 then describes the 3 prototypes that has been build, each using a different technology for adding redundancy to the system:

- A passive redundancy solution using so called distributed shared memory (DSM). This is build using the software product [Terracotta] (chapter 6.2).
- An active redundancy solution using network multicast communication. This is build using the software product [Hazelcast] (chapter 6.3).
- A solution using the so called end-to-end principle with "soft state" at the wind farm server nodes (chapter 6.4).

Chapter 7 describes a setup for testing the 3 prototypes in a realistic scenario with 18 nodes. As it is expensive and cumbersome to setup and maintain a test laboratory with 18 physical nodes, the test bench uses virtual machines using the Amazon Elastic Compute Cloud ([Amazon EC2]) computing platform.

Finally chapter 8 evaluates and compares the results of running the 3 prototypes on the Amazon EC2 test bench.

2 The Distributed Wind Farm

The system used as case for this thesis, is the distributed wind farm SCADA system briefly introduced in chapter 1.1. In this chapter the system is described in closer details by describing the functional requirements and the architecture of the existing system. After this the new architectural availability requirements for the system are described.

2.1 Functional Requirements

Focusing on the core monitoring functionality of the wind farm system, the central functional requirement of the system is:

> To monitor soft real-time values for a client specified list of sensors in a turbine. The soft deadline for the periodic readings is one second.

An elaboration of the terms used in this very concise sentence might be appropriate - the main goal is for clients (where "clients" can include other systems, as well as end user clients) to be able to monitor values from a list of turbine sensors. An example could be to monitor values for measured wind speed, power production, gear temperature, and so on.

The term *client specified list* refers to the fact that it must be possible for the clients to specify what sensor readings should be monitored. So the list of

monitored sensors will change over time depending on what information clients require.

The term *soft real-time* is used in the standard computer science way, meaning that the system should expect and be tolerant to occasionally missed deadlines and deadlines that are missed with a small time period. In the context of the wind farm system this means that the system is fulfilling this requirement even if there are occasionally two seconds between a reading, or if the period between two readings fluctuates around one second, sometimes exceeding the one second deadline with for instance 100 milliseconds.

This is opposed to a *hard real-time* system where the system has a fault if a single deadline is missed or exceeded. So the distinction between these two types of systems, is basically that in a soft real-time system a response or action that almost makes the deadline is still of at least some value, although the value may be decreasing after the deadline is surpassed, whereas in a hard real-time system the action or response is of no value if exceeding the deadline. The classical examples of systems with hard deadlines includes systems where human lives or health are at stake in case of failure, covering areas such as brake systems in cars, fly-by-wire control systems for aircrafts or security mechanisms in robots.

Finally the *deadline of one second* is actually depending on the latency and bandwidth of the client-to-wind-farm-server link as depicted on figure 3 on page 9. This means that for systems where the end-to-end system should be considered a soft real-time system the quality of this link must be evaluated before committing to the one second deadline - this is typically the case where the "client" is another system, doing for instance data acquisition or alarm monitoring. This should be compared to more ad hoc end user client connections over dial-in lines of varying quality - these are more to be seen as so called *interactive* clients instead of soft real-time clients (in the taxonomy of system response times described in [Burns & Wellings, 2001, ch.12.6] an *interactive* system is a system where there are no real deadlines, but the goal is just to strive for "adequate response times").

2.2 Architectural Description of the Existing System

To describe the architecture of the existing distributed system, the larger elements and the relationships between these are described below in a traditional viewpoint based way. As the architectural viewpoints the three viewpoints from [Christensen et al., 2007][3] has been selected:

Module: Mapping the functionality onto static development units.

Component and Connector (C & C): Mapping the functionality onto runtime components.

Allocation: Mapping the software entities onto the system "environment" (physical and logical hardware).

Since the intention of the architectural description in this case is to *communicate* information on specific parts and properties of the system, lots of information about the real system has been omitted to make the architectural structures regarding the monitoring part under analysis clear.

[3] Who bases their work on the recommendations of [Clements et al., 2003].

2.2.1 Component and Connector View

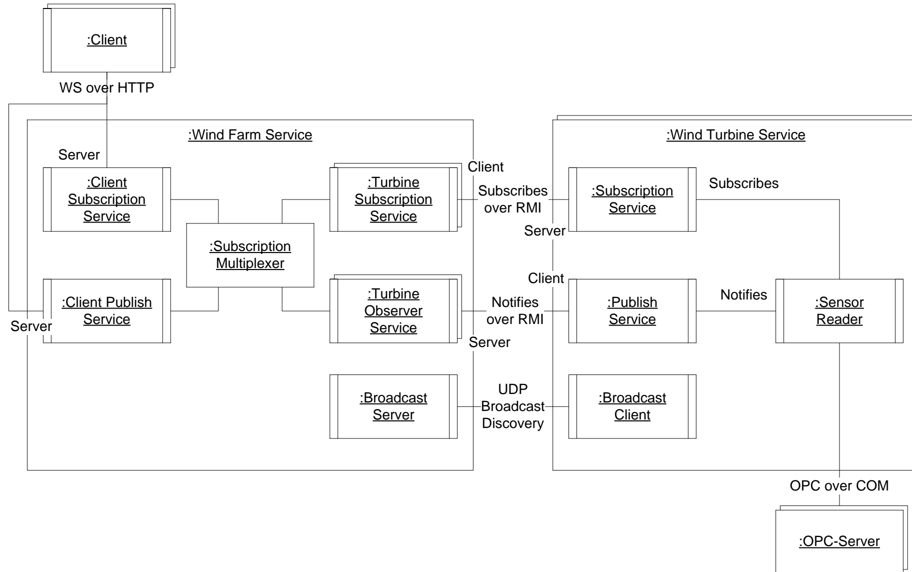


Figure 5: Overall component and connector diagram.

Figure 5 shows the overall component and connector diagram. The properties of the individual components and connectors are listed below.

Client: The client component. As the internal of the different client types are not central to this thesis, no decomposition of the different client types has been done. It should be noted though that all communication between clients and the *wind farm service* is initiated by the client. This includes the "publishing" of changed data from the *client publish service* on the *wind farm service* (i.e. due to the request-response properties of HTTP this is implemented as one second pollings of the *client publish service*, requesting changed data since last polling). This means that the so called flow-control is the responsibility of the client.

Wind Farm Service: This is the service running on the central node (as is later seen on the deployment diagram). This contains connections to all wind turbine services in the system, using Java RMI as the communication protocol, and connections to all clients, using web services over HTTP. It contains the components listed below.

Client Subscription Service: A component exposing a subscription web service used by clients to setup, change and teardown subscriptions for sensor readings. One subscription can span over sensors from several turbines. Each incoming client subscription change is forwarded to the *subscription multiplexer* component for processing, and a unique subscription token is returned to the client.

Subscription Multiplexer: The main stateful component on the *wind farm service*. This keeps track of all client subscriptions, and multiplexes the

subscription requests into only one active subscription for each *wind turbine service* - i.e. one for each turbine. This component is not active in itself, and its exposed data structures should be threadsafe as it is used from threads in several components (e.g. the *client subscription service* and *turbine observer service* writes data to this component, whereas the *turbine subscription service* and *client publish service* reads data from this component).

Turbine Subscription Service: This component regularly polls the subscription multiplexer for changes to the subscription for the turbine it handles. I.e. one instance of this component exists for each *wind turbine service* in the overall system. This is the RMI stub end (client) of the remote communication to the *subscription service* on one *wind turbine service*.

Turbine Observer Service: This is a RMI skeleton (server) that is notified via callbacks when there are changed values for one or more sensors on the *wind turbine service* it handles. So for each remote *wind turbine service* there is an instance pair consisting of one { *turbine subscription service* ; *turbine observer service* } handling communication to exactly one remote service. The API exposed by this service is coarse grained, meaning that batches of changed sensor values are received in one method call, instead of individual sensor readings. The changed values are then updated in the *subscription multiplexer* component.

Broadcast Server: A simple UDP server listening for UDP multicast messages to a configurable port on the so called *local segment multicast address* 224.0.0.1. Basically it just replies with a simple echo message whenever a client sends a broadcast message looking for a *wind farm service*.

Client Publish Service: This component also exposes a HTTP web service. This is used when clients polls for changed data using the subscription token received when the client requested a subscription using the *client subscription service*.

Wind Turbine Service: This is the service running on each of the wind turbine nodes in the system. This communicates with the central *wind farm service* using RMI. It contains the components listed next.

Subscription Service: Exposes remote functionality for the *turbine subscription service* on the *wind farm service* to manage sensor value subscriptions on the connected wind turbine. I.e. the public part of this component is exposed as an RMI skeleton (server).

Sensor Reader: This component contains logic for periodically scheduled reading of subscribed sensor variables. Communication to the actual turbine takes place using a standard protocol for industrial automation solutions, called OLE for Process Control - Data Access (OPC DA) [OPC DA, 2003]. As the OLE part of the name implies this standard uses Microsoft's binary Component Object Model (COM) protocol for interprocess communication. Changes in the read sensor values are send as notification callbacks to the *publish service*.

Publish Service: The RMI stub end (client) of the *turbine observer service* running on the *wind farm service*. Sensor value changes are sent as remote notifications by this component.

Broadcast Client: A simple UDP client that matches the *broadcast server* on the *wind farm service*. When the *wind turbine service* starts, this client sends local network segment broadcast messages looking for a *broadcast server*. When it receives a positive reply, it registers the different *wind turbine service* components at the server. And afterwards it sends periodically RMI keep alive calls to the server to detect an unavailable *wind farm service*, in which case it will fall back to sending broadcast messages again.

OPC-Server: A vendor specific implementation of the OPC DA standard for communication with the actual sensors in the turbine. This component is delivered by the vendor of the bus soft- and hardware used for communication in the turbine. So basically the OPC-server acts as a concentrator or gateway to the signals and events on the bus systems in the turbine.

As described above there is a requirement that the communication between the client component and the wind farm server is done using HTTP, which is strictly request-response based. This request-response requirement makes it impossible to implement the notifications from wind farm server to clients using the natural callback based protocol shown in figure 6.

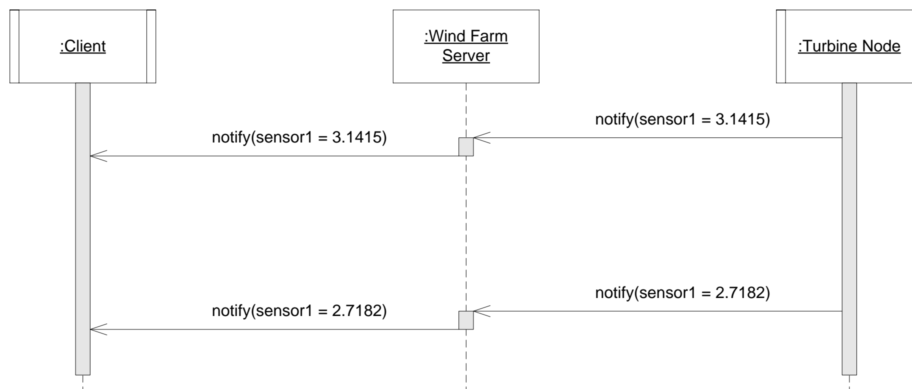


Figure 6: Natural implementation of event callback.
Not supported when HTTP is used.

Instead a poll-based event notification mechanism must be used to imitate the callbacks. This is implemented by letting the clients periodically poll the wind farm server for changes to the subscribed sensors. By letting the wind farm server only return the values for changed sensors since last poll, the data flow of the event based callback mechanism are imitated as closely as possibly given the underlying restraints of the request-response protocol. This mechanism is depicted in figure 7 on the next page.

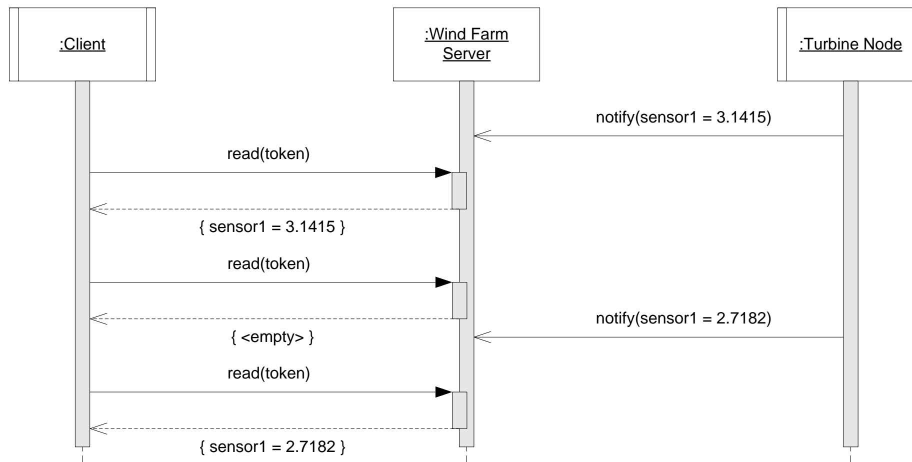


Figure 7: Actual implementation of event callbacks, using a poll-based mechanism.

2.2.2 Allocation View

The deployment diagram for the system is shown on figure 8. The components on this diagram should be easily recognizable from the previous C & C diagram. A few items might be worth a couple of additional words though.

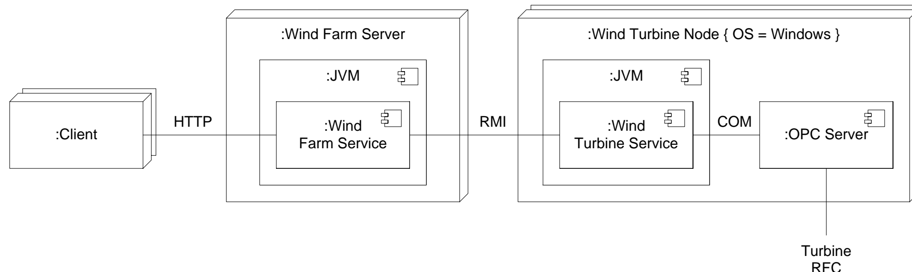


Figure 8: Deployment diagram.

Operating System: The operating system of the wind turbine nodes are restricted to Microsoft Windows due to the fact that the OPC DA specification is Windows centric as it is based upon Microsoft COM. Since the actual sensor readings are simulated inside the *wind turbine service* used in this thesis this restriction is not enforced in this work.

Turbine RFC: The remote field controller (RFC) is a hardware unit that runs a hard real-time operating system that controls the turbine. This is the center of all access to the communication bus running in the turbine. The communication between the RFC and the OPC-server is specific to the vendor of the RFC.

2.2.3 Module View

Most of the static development units (Java packages and classes) in the system are not vital to the understanding of the relevant parts of the system for this

thesis. The only part that it is necessary to understand is the data-model for subscriptions shown as a class diagram in figure 9. This shows the data model used in the *subscription multiplexer* component.

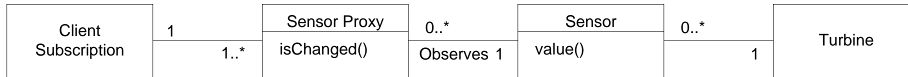


Figure 9: Class diagram for subscriptions in the *subscription multiplexer* component.

Basically this shows that one turbine has several sensors, and that one sensor can be observed by many client subscriptions. The only non-obvious class is the *sensor proxy* class, that has a "changed" state for a sensor. Each client subscription has its own copies of proxy classes for each observed sensor. This is necessary because of the "polled-based HTTP notifications" that takes place between the clients and the wind farm service, as this has the effect that a client subscription has its own independent view of whether a sensor has changed since its last reading.

This is necessary since the protocol between clients and the wind farm service are optimized to minimize the message payload size, such that each client poll for changed values only returns the actual changed values since last call. This could of course be criticised as an optimization that introduces additional client session state on the farm servers. But it is an optimization that is necessary due to the fact that some clients will connect over relatively slow telephone lines in form of single-channel ISDN lines, and will subscribe to sensor values for more than 100 sensors at the same time.

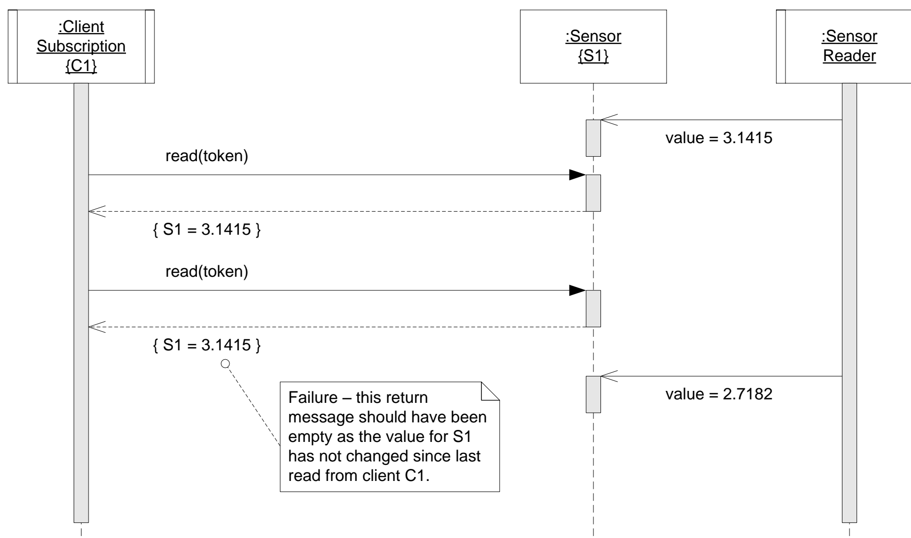


Figure 10: Sensor subscriptions - showing why the *sensor proxy* is necessary. Note that the "sensor" object is not the physical sensor, but the OO in-memory representation of the sensor. That is why the sensor reader updates this object with the values it has read from the physical sensor.

The sequence diagrams on figure 10 and figure 11 on the next page shows this mechanism. First in figure 10 it is depicted why this per client state requires that

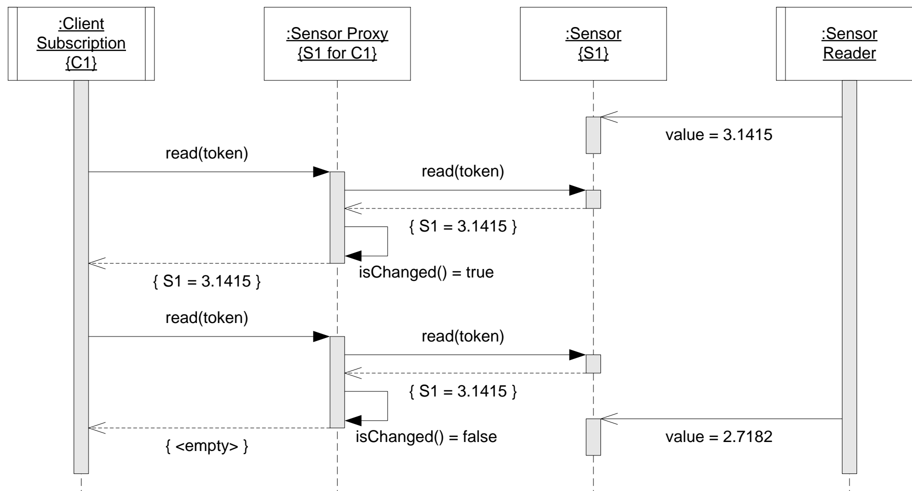


Figure 11: Sensor subscriptions with the *sensor proxy*. It is worth noting that the communication between the client and the sensor proxy is remote communication over HTTP, whereas the communication between the sensor proxy and the sensor is a standard local in-process method call.

the sensor proxies are present as the described use case is simply not supported without. Thereafter figure 11 shows the actual implementation with the sensor proxy objects.

2.3 The System Components in Client-Server Terms

To give the reader further understanding of the roles of the three distributed components, the *farm server*, the *turbine nodes* and the *clients*, it is preferable to think of each of these as participants in the traditional client-server relation. Furthermore this characterisation will help in fitting them into the relevant replication roles, which is the subject of chapter 3.5.

The focus in the description of the roles, is on the components relation to the state in the farm server, as this state is subject to later replication across a number of farm servers.

Farm server: The farm server acts as a ***server***, both in the relation to the turbine nodes and in the relation to the clients. The reader might object towards the role as a server in the relation to the turbine nodes, since it is the *turbine subscription service* on the farm server that actively calls the *subscription service* on the turbine node. To this it should be replied that the main stream of traffic in the relation, the delivery of sensor readings, goes from the turbine node towards the farm server. So the only reason that there are some traffic going in the other direction, is because the farm server and turbine nodes enters a long running relation when the turbine node registers itself at the farm server on startup. The roles of the farm server are, as briefly mentioned in the overview of the system in chapter 1.1, primarily to act as a middleman in the observer-pattern, where it takes the following roles:

- **Forwarding observer:** Conceptually it forwards the notifications of sensor changes from the turbine nodes to the clients.
- **Notification mailbox:** But implementation wise the forwarding is implemented in form of acting as a notification mailbox in the observer pattern until clients are ready to fetch the actual notifications.
- **Multiplexer/broadcaster:** It multiplexes several subscriptions for one particular sensor at one turbine node into only one actual subscription on each turbine node. So when a notification comes back from a turbine node, it broadcasts it to all the registered subscribers (i.e. clients). This mechanism isolates the load of the internal network from the number of end-user clients. I.e. the network notification traffic will not increase between the farm server and the turbine nodes, no matter how many clients subscribes to a given value. Figure 12 shows this multiplexing.

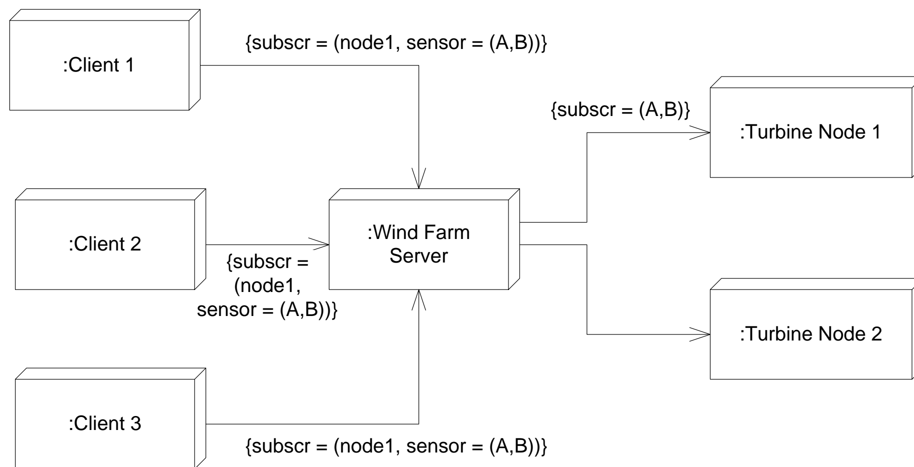


Figure 12: Multiplexing 3 client subscriptions for the same data into 1 subscription on the turbine node.

Turbine node: With respect to the state that should be replicated across farm servers, that is the sensor readings, the turbine nodes acts as ***write-only clients*** in the relation towards the farm server. In this it should be noted that as long as the communication channel from the turbine node to the farm server guarantees correct delivery order of the sensor reading messages sent from the turbine nodes, there are no synchronization problems, as the data delivered by one turbine node does not interfere with the data delivered by other turbine nodes (since they read different physical sensors). So with respect to the newest value for a sensor stored at the farm server each turbine node act as a single-writer.

Clients: The name of the client component gives a good hint as to its role, as it takes the role of a ***read-write client***. With respect to the ongoing modification of what sensors it subscribes to, it acts as a writer, and with respect to the polling for new sensor readings (i.e. notifications) from the

farm server it acts as a reader. For the clients it should be clear that there will be potential synchronization problems as the clients subscriptions spans over the same sensors, so the update of the data structures on the farm server that holds information about what clients are interested in events from a specific sensor, must be correctly synchronized. So the clients acts as multiple readers for the sensor readings, and multiple writers for the subscription data structures.

2.4 Distributed Data Structures

To further analyse and understand the problems with introducing redundancy in the existing system, it is important to understand what and how data structures are spread across several nodes in the system. These data structures are described in this section, and although this is an unusual architectural view, a view-based architectural description is more focused on making a clear and full description of a system, than following a strict dogmatic set of fixed views.

On the conceptual level, without going into the actual implementation of the data structure in Java, the data structures that spans over several nodes are basically only the data shown in figure 9 on page 18. I.e. the information on:

Relations: Representing what clients subscribes to what sensor readings. These will only change when client modifies their current subscription.

Sensor values: Current values for all currently observed sensors. These will change every time a turbine node delivers changed sensor readings.

Per subscription sensor values: The values per client subscription about whether a given sensor has changed since the last client polling for changed values, i.e. the information represented in the sensor proxy object in the figure. If this information is represented as a simple boolean flag, it will be raised every time the relevant sensor value changes, and it will be cleared every time the client polls for changed values.

A visual representation of these data structures across the components are depicted on figure 13 on the following page.

With respect to the *relations* and the current observed *sensor values* it should be clear that to reach a fully redundant solution this information must somehow be available on all copies of a running farm server.

With respect to the *per subscription information* represented in the sensor proxies, these are actually only relevant on a given farm server if the relevant client actually uses that particular farm server for polling for sensor values. And since this information is the most volatile in the system, it would be preferable if this information does not have to be replicated. This issue will be returned to in chapter 4.2 after the theory on replication has been described.

2.5 Architectural System Quality Attributes

Focusing specifically on evolving the existing wind farm SCADA system into a highly available system, the architectural drivers and system quality attribute scenarios that drives the refactoring are described in the two subsections below.

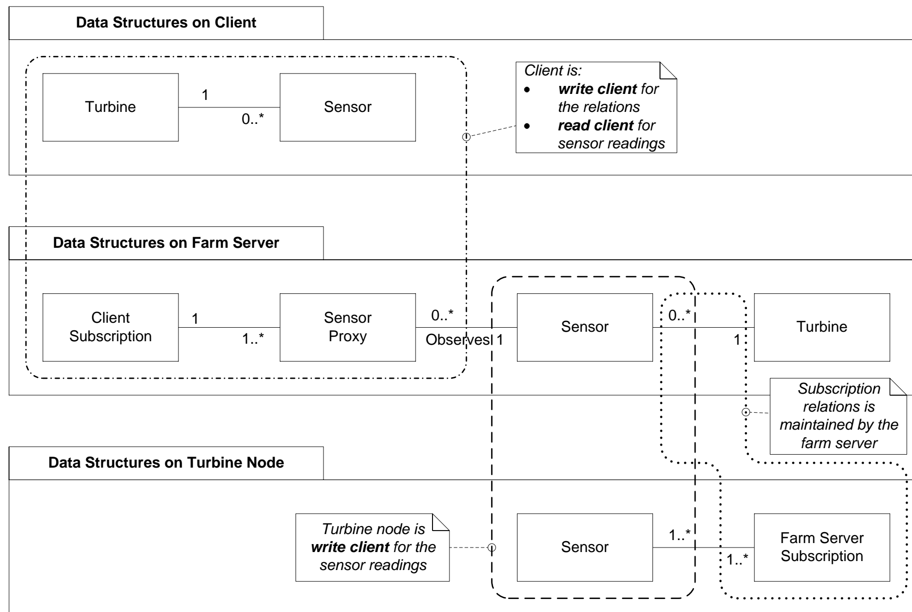


Figure 13: Distribution of shared data structures with comments on which component acts as reader or writer of the data.

2.5.1 Architectural Drivers

Availability: This is of course the main architectural driver, i.e. to change the architecture of the system to have a higher availability, where the system will continue to work even in the case of a hardware failure of the central wind farm server node.

Performance: Performance is not an explicit goal for the architectural changes, as the system as it is today has adequate performance. Nevertheless it is important not to lose focus on performance entirely, as quite a few availability solutions has an adverse effect on performance as additional overhead is introduced by these. In all fairness it should be said that in certain scenarios some availability tactics actually has a positive influence on performance when systems are running in normal operation. An example of this is replication of strictly immutable data.

2.5.2 Quality Attribute Scenarios (QAS)

The QAS's for these architectural drivers are described in the [Barbacci et al., 2003] format in the tables below.

<table>
<tr><th colspan="3">Scenario Refinement for QAS1</th></tr>
<tr><td colspan="2">Scenario(s):</td><td>An instance of the wind farm server crashes, this is detected by all clients using the server, and the clients automatically, without user intervention, switches to using another wind farm server. The period where no new data arrives at the client is at most 5 seconds.</td></tr>
<tr><td colspan="2">Relevant Quality Attributes:</td><td>Availability</td></tr>
<tr><td rowspan="6">Scenario Parts</td><td>Source:</td><td>Internally.</td></tr>
<tr><td>Stimulus:</td><td>A crash.</td></tr>
<tr><td>Artifact:</td><td>A wind farm server node.</td></tr>
<tr><td>Environment:</td><td>Normal operation.</td></tr>
<tr><td>Response:</td><td>The clients and turbine nodes detects that a wind farm server instance is crashed, and switches to use another wind farm server instance for communication.</td></tr>
<tr><td>Response Measure:</td><td>The flow of sensor readings should continue without stop, with a single delay at failover time that is at most 5 seconds long.</td></tr>
<tr><td colspan="2">Questions:</td><td>N/A.</td></tr>
<tr><td colspan="2">Issues:</td><td>For hanging servers (i.e. ”non-halt” crashes) the 5 seconds deadline will not be reachable as the typical TCP/IP timeouts will be longer than this, making it impossible to detect the crash within the 5 seconds window.

The failover is a protocol option to the client nodes. For some existing types of clients the cost of adding automatic failover is not economically feasible. These clients should continue to work as today.

The 5 second deadline can of course only be guaranteed for the clients where the connection between client and farm server is of a quality where the normal operation 1 second soft deadline for sensor readings applies.</td></tr>
</table>

Table 1: Quality attribute scenario 1.

Scenario Refinement for QAS2		
Scenario(s):		Several instances of the wind farm server are unavailable. The scenario described in QAS1 must hold for the case where 2 wind farm servers are down at the same time.
Relevant Quality Attributes:		Availability
Scenario Parts	**Source:**	Internally.
	Stimulus:	Two crashes.
	Artifact:	Two wind farm servers crashes (i.e. are down at the same point in time).
	Environment:	Normal operation.
	Response:	As QAS1
	Response Measure:	As QAS1
Questions:		N/A
Issues:		It is implicitly given by this QAS that the system will not be able to operate in normal mode if *more* than 2 wind farm servers are unavailable at the same point in time.

Table 2: Quality attribute scenario 2.

Scenario Refinement for QAS3		
Scenario(s):		The system should be be able to deliver sensor readings every second for 500 sensors distributed over 10 turbines to 5 concurrent clients, where each client subscribes to values from 100 of these sensors.
Relevant Quality Attributes:		Performance
Scenario Parts	**Source:**	5 concurrent clients.
	Stimulus:	Each client subscribes to 100 sensors distributed over 10 different turbine nodes.
	Artifact:	System.
	Environment:	Normal operation.
	Response:	Clients receives sensor readings as they change.
	Response Measure:	The sensor readings arrives with 1 second intervals.
Questions:		N/A
Issues:		The 1 second deadline for sensor readings is a soft deadline, meaning that it may occasionally be missed or be delayed. Furthermore it only applies for clients where the connection between client and farm server is of a sufficient quality.

Table 3: Quality attribute scenario 3.

A few comments on some of these QASes seems appropriate. First and foremost the reader will probably have noticed that quite a few sentences have been used to describe in very vague terms that there are some requirements for the communication line between the clients and the wind farm server, and that this must be of ".. *sufficient quality*". This vague term is of course not satisfying for a QAS that should in its nature be as clear and unambiguous as possible. Unfortunately it is the case for the wind farm SCADA system that the network connectivity from clients varies over a broad range, spanning from single line ISDN connections of low quality at the low end of the spectrum to high-quality fixed connections at the high end. Therefore this loophole exist to be able to not promise performance that will never fly for the low end of the spectrum.

Of course the vague terms should in the perfect world be exchanged with more precise wordings describing the minimum requirements to the communication lines where these QASes will apply. Words that should describe the requirements with respect to bandwidth and latency. But although it should be possible to find these requirements through tests and measurements at various line types this is work for another time. For the prototypes created as part of this thesis it suffices to say that these will work with communication lines of ".. *sufficient quality*" to fulfill the performance QASes.

2.6 Source Code for the System

At the offset of the thesis work, it was the intention to build the prototypes directly upon the existing system and measure directly on these with respect to fulfillment of the QASes. Unfortunately the existing system contains a lot more functionality, complexity and network traffic that are more business related and are not really related to the availability of the system, and are therefore outside of the focus of this thesis.

Therefore it was clear that to get a manageable and clean baseline system to build upon, it would be necessary to extract the relevant components from the existing system. Unfortunately the components in this system has undergone several revisions and do no longer exhibit a particular low coupling between components[4]. So after having battled with this, it was deemed easier to make a complete re-implementation of the relevant parts for the model together with a simple mock implementation of the sensors, making it possible to run the tests and scale the test system without having to be limited by actual hardware RFC resources. Furthermore this makes it possible to loosen the operating system restriction for the wind turbine nodes, since the OPC-server requiring Windows is not necessary on the test systems when no real RFC communication takes place.

The model system of course fulfills the performance QAS 3, but does not fulfill the availability QASes 1 and 2.

Appendix C gives an overview of how the architectural description in the previous chapters maps to the source code in the model system.

[4]This seems to be a failure to fulfill standard good practices with respect to architectural *modifiability* quality attributes. This might be the case, but in hindsight without further information, it is hard to make that conclusion, as *modifiability* might never have been a prioritized architectural quality attribute.

3 Availability Theory

To change the existing system to fulfill the new availability QASes it is necessary to be familiar with the computer science theory on the end-to-end principle, failure semantics, replication and architectural tactics for introducing availability properties into a system. This chapter gives an introduction to these areas for the reader for whom these theories are somewhat unfamiliar. For other readers this chapter will contain well-known material and can probably be read fast by skimming.

3.1 Availability - The Uptime Percentage

The standard way to look upon availability of a system, is to use the *uptime percentage* as goal. Like for instance the famous 5-nines goal 99,999%. In the context of the wind farm SCADA system, it should in theory be possible to calculate such an availability percentage for the wind farm servers following the traditional formula:

$$1 - p^n$$

Where p is the probability of one wind farm server being unavailable, and n is the number of wind farm servers. I.e. this expression calculates the probability of at least one of the central servers being available.

This means that for a system where $p = 5\%$ the overall availability will be increased from 95% to 99.75% just by introducing one independent replicate of the central wind farm server.

The observant reader will of course have noticed that this probability is only the probability of *one of the wind farm servers being available*, which only serves as one factor in calculating the overall system uptime.

So measuring uptime percentages is all well and good, but nevertheless it should be noted that none of the previous availability QASes uses an uptime percentage as success criteria. This is by intention as it seen from a technical point of view is simple impossible to *prove* upfront that a complex system with a multitude of different cooperating hardware and software components like the wind farm SCADA system has an availability of X percentage.

So for the SCADA system it is much more measurable to specify the availability goal as *"can or can we not survive two concurrently down farm server nodes"*, as QAS1 and QAS2 describes. The uptime percentage is then better left for the system level agreements (SLA) and the political and non-technical battles related to that.

3.2 The End-to-End Principle

If a system is build using stateless network nodes and fulfills the so called *end-to-end principle* as described in [Saltzer et al., 1984] and elaborated on in [RFC 3724], it will greatly simplify the task of adding redundancy to a system, as it can mainly be done by adding additional redundant stateless network nodes. It should therefore be considered whether a distributed system where high availability is required fulfills this principle.

In brief the *end-to-end principle* when applied to network nodes states that:

> The network itself should not contain any state, leading to the fate of a conversation to only depend upon the endpoints. And in the case where state is maintained in network hops along the route between the endpoints, this state should be minimized and self-healing, so that the conversational state can be reconstructed in case a network node crashes.

In the lingo used in [RFC 3724] the reconstructable state is labelled “soft state”, whereas the critical conversational state that should only be maintained in the endpoints is labelled “hard state”. In the wind farm SCADA system the central farm server should be seen as a network hop between the endpoints nodes, the clients and wind turbine nodes.

3.3 Failure Semantics Theory

To be able to choose the architectural tactics for solving the availability problems in the wind farm system, it is necessary to delve into the theory on what kind of faults can arise in the system. A very thorough analysis of this area of so called *failure semantics* in context of distributed systems is found in [Cristian, 1993]. Although the paper is a little dated, and a lot of the examples in the paper uses hardware of a foregone era[5] as case-studies, the analysis of the different fault types and architectural remedies for handling them is still relevant.

The reader should pay attention to the fact that Cristian and his contemporaries like [McAllister & Vouk, 1996] does not have a clear distinction between *faults* and *failures.* In this thesis these terms will be used like in the later works of [Burns & Wellings, 2001] and [Bass et al., 2003], where *failure* refers to externally observable system failures triggered by component *faults* that have not been masked or corrected inside some system component. In a system with interacting components it should be clear though that the failures of one component, is actually faults when seen from a dependent component, so the separation into these two terms can quickly become a battle of words.

Cristian classifies component failures into three main groups:

Value failure: That an output value of a component is wrong[6].

Time failure: That the timing of a response is incorrect.

Arbitrary failure: A combination of a value and a time failure, e.g. an incorrect response delivered to late.

Cristian describes several subclasses of these main groups:

- Crash Failures: The situation where a component completely stops handling requests.
- Omission Failures: The situation where a component does not respond to one or more requests, but the component has not crashed, and will continue to answer other or later requests.

[5] It is probably not many persons under the age of 35 that has even heard of the DEC VAX or the Tandem systems that are described in the paper.

[6] Note Cristian labels this group *response* failures.

- Corruption Failures: Where a response of a component is incorrect. This incorrect response can both be incorrect with respect to a returned value or with respect to an incorrect internal state transition.
- Timing Failures: Can be split into both to late or to early responses.

3.3.1 Choosing the Relevant Faults

The challenge for an architect is then two fold; first to find out exactly what kind of faults can take place in the system, and how they should be handled. The QASes in chapter 2.5.2 should act as the guideline for this, as these should highlight the availability scenarios the different system stakeholders find of most importance.

A different school of thought, which probably originates from the physical hardware engineers, is presented by trying to measure actual reliability metrics. [Sommerville, 2007, ch.9.4] describes a list of these with the well known mean-time-to-failure (MTTF) as one.

Along these lines Cristian points out that the correct way to characterise a specific fault as negligible is to base this judgment on hard measurements and a stochastic probability distribution for the fault.

Although these recommendations seen from a theoretical point of view are correct, it should be noted that these hard low-level measurable metrics are not that relevant in a large distributed system like the wind farm SCADA system, where standard commodity hardware is used in setups that are individual tailored to each wind farm. An example of this is the simple fact that the physical layout of two wind farms are not identical as this depends on the topography of each site. This of course leads to variations in things like cable lengths, leading to different latency characteristics for each wind farm. Given complexities like this and the low volume of installations, it is simply not economically feasible to strive for calculating a theoretically perfect probability distribution when solving availability problems for systems like the wind farm system. This follows of course along the lines of the previous discussion of dismissing the SLA uptime percentage as more of a political than a technical measure when talking about complex distributed systems like the wind farm system.

3.3.2 Handling Faults - Mask, Propagate or Correct

Based on the decision of what faults the architecture should be robust with respect to, the second challenge for the architect is then to find architectural solutions that will either mask or correct the faults as they arise. The actual solutions, in form of architectural tactics are described in chapter 3.4, whereas the current chapter describes the different approaches of handling faults from a higher perspective.

Seen from the individual system component where a fault is triggered, there are three ways to deal with it:

Masked: The component can mask the fault so the users of the component will never observe an external component failure, even though the component or part of it is still in an erroneous or degraded mode.

Corrected/recovered: The fault can be fully corrected, i.e. full recovery of the fault will take place in the component itself. The corrected fault will of course not be observable from the outside.

Propagated: The component can "give up" and propagate the fault, whereby it will turn into an external observable component failure.

It should of course be mentioned that the *propagated* solution does not actually handle the fault. But as a system typically consists of many components and layers of components, this will not necessarily lead to an overall external system failure, as the individual component failure can be dealt with in a combined effort between several components, or it can be handled at a higher level. These two solutions are labelled by Cristian as *group masking* and *hierarchical masking*.

3.3.3 Cooperative Masking: Hierarchical

Hierarchical masking is what takes place when a higher level component handles the failure in some way. One typical hierarchical solution is to replay the failing request towards the failing component (in a typical database bound system, an example of this could be by re-issuing a transaction that has been aborted by the database manager). Solutions for hierarchical masking therefore often takes the form of exception handling code in the components calling the failing component.

It should be obvious that making strong fault resistant components at a low level will of course make it easier to implement the higher level components, as they do not have to deal with as many fault scenarios. But as high fault resistance at a low level does not come without a cost, the pros and cons of this must be balanced pragmatically as described in the classical article by Saltzer et al. on the *end-to-end principle* ([Saltzer et al., 1984]).

So as it is often the case in engineering disciplines there are no one-solution-fits-all - as an example of this one can consider the transport layer in the IP-stack, where both TCP with all its reliability guarantees is available, as well as UDP which gives no delivery or ordering guarantees. In some scenarios it is a better overall solution to choose the much cheaper UDP (cheaper in cost of messages and overhead) for communication, even though it may require additional error handling in a higher level component.

3.3.4 Cooperative Masking: Group

The other type of cooperative masking is called *group masking* where a number of components form a group, hiding failures in any individual group member from outside users by the group management mechanisms. In a pure group masking solution the users of the group does not see the individual members, as these are abstracted away by the group management mechanism. What the client sees is just the output of the entire group, which is of course a function of the outputs of the individual group members.

Figure 14 on the next page shows the typical way to hide the individual group members from the clients using the group, using a so called group *front end (FE)* component. The reader should not necessarily focus on where the FE is placed on the figure, but more focus on the mechanism. In some solutions the FEs are in themselves nodes in a distributed systems, and in other solutions

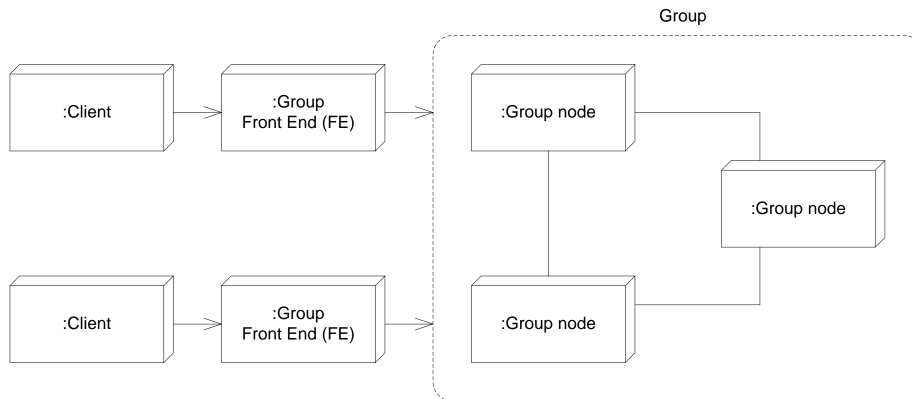


Figure 14: The role of the front end (FE) components in group masking.

the FEs are actually libraries linked into the client, whereby they are actually running in the client process space.

Exactly how the group output function is designed also differs from solution to solution. The design should take into account that the *failure semantics of the entire group* depends both on the individual member nodes failure semantics as well as the failure semantics of the communication mechanisms between the nodes. If both of these exhibit strong guarantees with respect to the failure semantics, e.g. by guaranteeing that there will be no corruption faults neither from the components, nor from the message transport on "the wire" between components, the group output function can simply be: "Use the first response that is received from any group member". If on the other hand either the nodes or the communication mechanism can not give this kind of guarantee, the group function instead has to be designed to use voting, in which case replies from a majority of the nodes must be awaited before the group function can reply.

The final large question for group masking is what *group size* should be used. Again this depends on what failure semantics can be guaranteed in the group. Furthermore it of course depends on what kind of tolerance must be achieved. For this Cristian uses the term *K-fault tolerance*, where K is the maximum number of failing nodes the group should be able to handle[7]. In the most general case where a group has to deal with arbitrary or corruption failures the group size has to be $3K+1$ as proved in the [Lamport et al., 1982] classical article on the *Byzantine Generals Problem*. If the group instead should only handle crash and omission failures a group size of $K+1$ should be sufficient. But generally it can be said that a large number of group members leads to a high K, but with the cost of higher communication overhead. The exact opposite properties are of course experienced for groups with a small number of members.

As a closing comment on the failure semantics, it should once more be stressed that since we are discussing distributed systems, the most important fact to remember, is that for both hierarchical and group masking solutions the actual observed failure semantic of a lower level component is a function of *both* the failure semantic of the actual component *and* the failure semantic of the communication layer used for communication with the component.

[7] $K = 1$ is of course a so called single fault tolerant group.

3.4 Tactics for Increasing Availability

After having described the theory on failure semantics and the abstract theoretical ways of handling faults in a system, it is time for turning towards the actual architectural tactics for increasing availability.

A list of the overall types of architectural tactics for this is described in [Bass et al., 2003, ch.5.2]. In the following paragraphs each of the four tactics are shortly presented, before the suitability of each individual tactic, if applied to the wind farm SCADA system, is later analysed in chapter 4.2 on page 42.

3.4.1 Voting

The *voting* tactic is a rather expensive tactic, that is really only suitable in systems where corruption faults must be handled. Faults that can either stem from defective components and/or from transmission failures.

With respect to corruption faults introduced by the transmission mechanisms, these are relatively easy to abstract away on the higher levels by using some kind of CRC-codes at the lower levels combined with appropriate retransmissions. Which is in practice what happens when a reliable transmission protocol like TCP is used.

With respect to corruption faults introduced by errors in either hardware or software, along the component path all the way from the sensor readings to the client, these are a lot harder to handle at lower levels. So in this case the decision for ignoring these kind of errors, is instead based on pure "economic" considerations. I.e. the voting tactics for handling this, no matter whether a parallel group masking solution in form of so called N-version programming, or a sequential hierarchical masking solution in form of recovery blocks, are simply to expensive to implement.

For completeness it should be mentioned that a hierarchical solution with recovery blocks can not detect and handle all kinds of corruption faults as well as a group solution with N-version programming can. In a recovery block solution, faults can only be handled if it is possible to detect that a corruption fault has happened in a given block. So for certain systems only a N-version solution will make sense. For the interested reader [Burns & Wellings, 2001, ch.5.4-5.5] contains in depth coverage of these two kinds of voting tactics, and also gives a thorough listing of mechanisms for actually detecting some kind of corruption faults in a recovery block solution.

Burns & Wellings furthermore describes the two above tactics as respectively *static redundancy* and *dynamic redundancy*:

Static redundancy: In a static redundancy solution all the components are used on each request. I.e. *statically* no matter whether faults occur. N-version programming is god example of this.

Dynamic redundancy: Opposite to the static redundancy, solutions with dynamic redundancy has the properties that the components are used *dynamically* only when actually required as faults occur. Recovery block programming fails into this group.

3.4.2 Spare

This is the solution where a cold spare machine is kept available for booting when required for replacing the central node. This is actually the state in the wind farm solution as it is today, which is deemed insufficient to fulfill the QASes of the wind farm SCADA system.

Following the introduction of the spare into the system, all the client subscriptions have to be set up again by each client.

3.4.3 Active Redundancy

Active redundancy as described by [Bass et al., 2003] is another type of *static redundancy*. It is defined as a system where the client sends the same request to all the redundant components in parallel, and they all process the request and replies. What distinguishes this from the voting tactic is that the client just uses the first response it receives and throws away all later responses to the same request.

This tactic as well as the N-version voting tactic should be easily recognisable as two different implementations of the so called *group output function* described on a higher theoretical level in chapter 3.3.2.

As a benefit of active redundancy, it is implicitly given that it might have a positive performance effect due to the implicit architectural tactic of "maintaining multiple copies of either data or computations". Unfortunately this must be weighted against the possible adverse performance effect due to the complexity and increased number of messages involved in keeping the multiple redundant components in synchronization.

3.4.4 Passive Redundancy

Passive redundancy is of the *dynamic redundancy* type, where the client only communicates with the primary component, falling back to one or more standby components as necessary if the primary fails. At takeover time the passive component must then be synchronized with the latest persisted state of the primary component, before being brought into service.

This type of redundancy opens up for a traditional computer science tradeoff, where it is possible to adjust how much the standby components are allowed to be "lagged", thereby trading lower messaging overhead at normal operation with a longer take over period as more state has to be synchronized.

The pros and cons for respectively active and passive redundancy are summarised in table 4 on the next page.

Whether an active or a passive redundancy solutions is used, the main question is, exactly what mechanisms are to be used for *replicating* the states between the redundant components. This will be the subject of the following chapter on replication theory.

3.5 Replication Theory and Tradeoffs

When surveying the theoretical landscape on *replication* it is important to keep in mind that the *goal* of introducing redundancy in a system is typically either to accomplish *high availability* of a system, to introduce *fault tolerance* in a

	Pros	Cons
Active Redundancy	Fast responses, even in the face of a failing component. All nodes are equal, so no election algorithm required.	Large overhead in exchanged network messages, as all components must be kept in "clock-step" synchronization at all times. Potential complex messaging protocols required to guarantee the global state synchronization.
Passive Redundancy	Smaller overhead in number of exchanged messages. Simpler message protocols as some state lag is allowed in certain time frames.	Response time will be slower when a failover between the primary and a standby component takes place. Requires a failure detection mechanism and an election algorithm for selecting the primary node.

Table 4: Pros and cons for active and passive redundancy.

system or to increase *performance* of a system. In general these three goals have slightly different properties with respect to the data consistency:

High availability: In redundant systems designed with this as the main goal, data might be out-of-date, i.e. lagged to a certain degree. The most important property is that data is always available.

Fault tolerant: In redundant systems designed with this as the main goal, the most important property is that data must always be correct.

Performance: In some systems data is replicated with the focus of increased overall system performance. This can both be in form of caching of the same data in the different *layers* in the system, or in form of introducing multiple copies of a node with the same data with the main goal of increasing performance, e.g. in form of geographical separation of the nodes, so the node used is always geographically close to the using systems. A good example of a geographically distributed system designed with performance as the goal, is the Akamai world wide network of servers for serving web-content.

It is important to keep these different types of goals in mind when analysing the replication theory as they have different impact on what replication properties are required and which one are optional. Note thought that these goals are not necessarily orthogonal. In some type of systems several of them are required. For

instance in the wind farm SCADA system, where some replication has already been introduced, in terms of caching sensor readings from the turbine nodes on the farm server, partly because of performance. It is a little ironical that this exact replication of data between farm server and turbine nodes partly to increase performance, is exactly what makes it harder for us to introduce replication of data *across* redundant farm servers.

As a quick side note, it might be necessary to address the fact that some readers will at first look have a strong natural resistance against the view that a high available system with inconsistent data is of any value at all. A resistance that can be phrased into a slogan like:

> *"If a system can not deliver correct data, it doesn't matter how fast it performs"*

On the surface this expression seems like a natural statement that holds in all cases. But as soon as one delves below the surface, it is clear that there are lots of systems where the *"correctness of data"* can be bent slightly and still comply to the business requirements of the system. A fact that is especially valid if the benefit of doing so, is a large performance increase.

It is of course important to state that when discussing systems allowing inconsistent data, we are talking about setting boundaries on the sort of inconsistency that are allowed. So it is not a about allowing *all* inconsistencies but only the *bounded range* of inconsistencies that makes sense in the given system.

3.5.1 Types of Replication Messages

When considering how data should be replicated between redundant nodes, it is important to be aware that depending on what type of replication messages are used, the requirements on the failure semantics offered by the communication layer, differs significantly.

It is therefore important to distinguish between the two following types of replication messages:

State: Replication systems sending only *full states* (i.e. data) in the communication, does not require a reliable communication layer. If a message is lost, it can just be resend, and receiving the same message twice is not a problem as the messages are omnipotent.

Operations on state: Whereas systems sending only *operations* on the data, requires a reliable communication layer. This is necessary since for the below equation to hold all operations must reach their destination once and exactly once:

$$current\,state = initial\,state + all\,operations$$

Coulouris et al. points out that the model with *operations on state* is equivalent to the typical message types used in client-server systems, where the servers owns the data[8] and offers methods for modifying them ([Coulouris et al., 2005, p.750]).

[8]The data can manifest itself in many ways, for example in terms of state, objects (encapsulating state) or resources (e.g. as controlling specific hardware, such as a printer or a disk-system).

3.5.2 Group Communication - Multicast Messages

When state or operations on state must be replicated on a group of nodes some kind of group communication is of course necessary. But when a message in a distributed system should be delivered to several nodes the communication is no longer as "simple" as when using normal one-to-one communication. The four main questions in group communication are:

Group membership: This question goes on what nodes in a system are actually part of the group at a specific time, i.e. to what nodes the messages should be delivered. The theory on this area is highly simplified if so called *static groups* are used, i.e. groups where members does not dynamically enters and leaves the group. In a static group all the members are there from the beginning and until the end, where the only option is that a member may crash and stop sending and receiving messages. The reader interested in dynamic groups with so called group views should consult the overview of the theory on this in for example [Coulouris et al., 2005, ch.15.2] or [Birman, 2005, ch.15].

Multicast reliability: The standard delivery guarantees that is given in one-to-one communication when using a protocol like TCP, is a lot harder to achieve when multicasts are used. So to be able to guarantee that if one member gets a message, all other members gets the same message, relatively complex protocols must be used. [Coulouris et al., 2005, ch.12.4] gives an overview of the theory on this area.

Message ordering: Even if reliable multicast is implemented, there is still no guarantees as to the order of the messages on the individual member nodes. Messages sent from different members can be seen in entirely arbitrary order at the other group members. Some ordering guarantees can be given by applying yet another protocol on top of the protocol used for reliable multicast. A protocol that typically guarantees one of the following message orderings[9]:

- **Total:** Defined as: *If a correct process delivers message m before it delivers m', then any other correct process that delivers m' will deliver m before m'.*
- **FIFO:** Defined as: *If a correct process issues multicast(m) and then multicast(m'), then every correct process that delivers m' will deliver m before m'.*
- **Causal:** Defined as: *If multicast(m) → multicast(m'), where → is the happened-before relation induced only by messages sent between the members of the group, then any correct process that delivers m' will deliver m before m'.*

These definitions might sound very much like alike, but as usual the devil is in the detail of the exact phrasing. Overall it can be said that the total ordering, is the only one of the three that gives a global ordering (i.e. the same message order on all members). Opposed to this the FIFO ordering only describes the ordering of messages sent by one process. And finally

[9]Definitions taken from [Coulouris et al., 2005, ch.12.4.3]

the causal ordering builds upon the classical [Lamport, 1978] *happened-before relation* for ordering some but not all events in a distributed system. For a global ordering with the best event ordering that can be achieved in a distributed system, the total and causal ordering can be combined. Finally it should be noted that any multicast protocol that fulfills causal ordering also trivial fulfills FIFO ordering.

Elections: In the passive redundancy tactic, and in some algorithms for handling dynamic groups it is required that one of the members of a group takes a special role, e.g. as a master in the group for failure detection. The same goes for some ordering algorithms, e.g. some of the protocols used for implementing total ordering, where a global sequencer node is required. In any of these cases an election algorithm is also required. The interested reader is referred to other work on the specialised topic on different election algorithms.

With reliable and ordered multicast it should be possible to build a replication version of the wind farm SCADA system, with either active or passive redundancy. But just as multicast is an abstraction build upon normal point-to-point communication, it is also possible to build further abstractions on top of multicast and see the replicated nodes as sharing one large virtual memory area, where a value written to a memory position by one node, will be atomically available for reading by all the other nodes. This is one incarnation of the distributed shared memory (DSM) model, and the topic of what kind of consistency guarantees can be given in such as system is the subject of the next chapter.

3.5.3 Memory Consistency Models

No matter whether a replicated system is build using a DSM middleware layer, build with some other middleware layer, or build manually using either low level point-to-point or multicast communication, the theory on *memory consistency models* is important to understand, as it highlights the different possibilities that generally exists in all types of system where

two or more processes accesses shared memory in some incarnation

This phrase is highlighted as it requires a little further explanations for one to be able to acknowledge how many types of systems the theory of memory consistency actually spans over:

- The term *processes* refers to any kind of computing processing that can take place at separate processing units. So the memory consistency models are relevant both in systems where the processes is running on two CPUs in one physical machine sharing the same physical RAM over a shared bus, as well as in systems where processes are running on two distinct physical machines where the shared memory is only virtual in terms of a DSM layer[10].
- The price of implementing strict memory consistency is of course higher and more complex in a distributed system, than in a system where all

[10]As a note it can be mentioned that even virtual machines, like the JVM requires and has well defined memory consistency models.

the processing takes place in the same physical machine. But the theories applies nevertheless to all systems whether they are in either end of the spectrum, or somewhere in between like for instance NUMA[11] systems.

In systems with shared memory the problem is that whenever a process reads a given memory location, there are actually very few guarantees that can be given to what value it will see. So although the layman will say that this is simple, it must be the last value written by any process, it is unfortunately not that simple as shown on figure 15 where the notation used is:

- W_l^v a *write* of the value v to location l.
- R_l^v a *read* of the location l returning the value v.

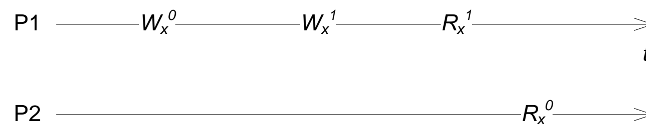


Figure 15: Memory consistency example with two processes P1 and P2 reading and writing to the same shared location x.

Although from the layman perspective the situation on the figure is clearly wrong, that the read performed by process P2 of the x-location will return a value, 0, long ago over written with a 1 by process P1, and even worse, a read that takes place *after* the read in process P1 has returned the "correct" value 1. A situation like this is unfortunately very well possible to see in a system, due to a lot of factors. In a distributed system, this can for example be due to latency, i.e. meaning that the effect of W_x^1 has not yet reached the cached location where P2 is reading from. In a single machine system, this can for instance be due to temporary CPU register caching or CPU level 1 or 2 caching of the location x in either process P1 or P2.

> *So the goal of defining an appropriate memory consistency model is to make it possible for the layers build upon the given system to be able to get guarantees about exactly how memory reads and writes will behave with respect to ordering of values read and written by the other processes in the system.*

The memory consistency model can in other words be seen as a sort of guaranteed *failure semantics* for the memory in a system. And as previously described a strong failure semantics at a lower level makes it easier to build the higher layers, but typically at increased costs at the lower levels. This is also exactly the case with respect to the memory model, where a strict consistency model will make it easier to build compilers and programs on the system, but have as a cost that *all* memory reads and writes are more expensive than in a more weak consistency model.

Furthermore it should be noted that the theory on memory consistency models is also used in relation to the computer science synchronization theory, where

[11] *Non uniform memory access* (NUMA) describes a conceptual type of system where there are different access times for different parts of the memory.

synchronization is both about atomicity of a sequence of operations in terms of ensuring *mutual exclusion* to so called *critical sections*, as well as *condition synchronization* or *memory visibility* between processes and threads.

[Mosberger, 1993] does a good job on describing the different theories that has been put forth on memory consistency models with different guarantees. A short distillation of the most relevant theories is described below in decreasing strictness. For each of these Mosberger includes examples of legal execution histories similar to the one in figure 15. The reader for whom memory consistency is a new concept, is encouraged to study these as they give a good understanding of the subtle differences between what the different models guarantee. The examples could of course have been included in this thesis, but since the Mosberger paper is so clear, the inclusion of the examples here would only have been a shallow copy of a clearly written paper.

Atomic consistency/linearizability: This is the most strict consistency model, where all reads and writes are serialized according to the fictional "real time", as if the entire system was one processing unit. For anything but actual single processor systems, this consistency model is usually to strict and expensive, but it typically acts as the baseline against which the weaker consistency models are compared.

Sequential consistency: This consistency model can be seen as an implementation of the atomic consistency model with a global ordering. But instead of the "real time" constraint, the global ordering must be consistent with the clause "*the operations of each individual processor appear in this sequence in the order specified by its program*".

Causal consistency: This takes the standard happened-before relation into account. Meaning that only reads and writes that are causally related are ordered, whereas causally unrelated events can be observed in different orders in the different processes. The reasons for this consistency model should be clear to anyone familiar with the concepts described in [Lamport, 1978].

Coherence: Where the previous consistency models all strives for some global ordering of the read and write events, the coherence model weakens this by only requiring sequential ordering on a *per-location* basis. So in a system consistent with coherence the different processes might see the writes to different locations in different orderings.

Pipelined RAM: This model is based upon the natural physical model that all reads from a processor occurs as reads of the locally cached memory value[12], and writes occurs by updating the locally cached value and broadcasting the written value to all the other processors. This has as consequence that all the write operations from a single processor is seen in the same order on all other nodes, but the intertwining of writes from different processors might be seen as different on different processors.

The above models are known as so called *uniform* consistency models. This is opposed to so called *hybrid* models where the *access type* or *context* in which the

[12]Whether it be in form of the local physical RAM in case of a DSM system, or in form of the level 1 or 2 cache in case of a system with shared physical RAM.

read or write operations are performed, is taken into account. The most used type of access to take into account in the models is some type of synchronization context for the operation, meaning that these models are only consistent, as long as the users respect the synchronization model. Three *hybrid* models described in [Mosberger, 1993] are:

Weak consistency: The most strict of the hybrid models, where all synchronization operations such as *acquire(lock)* and *release(lock)* are sequential consistent, and all reads and writes are globally ordered according to the *fence* operations that the synchronization operations actually represents. This makes it possible to only flush writes to all the other processors when a fence operation occur. Figure 16 gives a visual representation of the concept of fence operations, also known as memory barriers. Letting the user control when a fence operation takes place, of course puts more burden on the users of the system, whether it be a compiler or a standard program, as the memory accesses are only consistent when the synchronization operations are performed correctly at all processors.

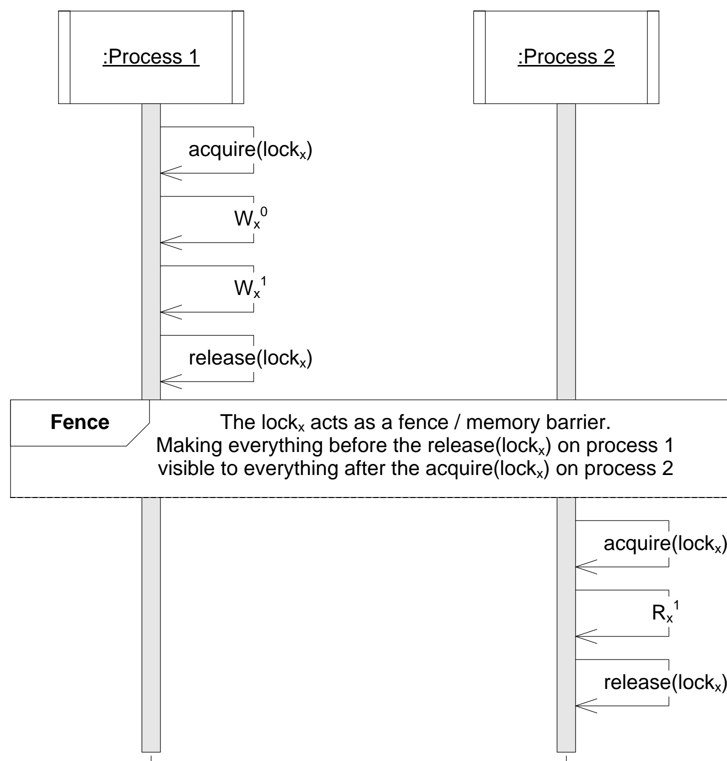


Figure 16: Visualization of the concept of *fence* operations.

Release consistency: This is a specialisation of the weak consistency model, where the different synchronization primitives has different influences on when written memory is flushed to the other processors.

Entry consistency: Is even weaker as instead of global fences it let the user of the system bind individual locations to different synchronization locks. So this can be seen as equivalent to the *per-location* basis uniform models. Furthermore it lets the user define the individual accesses into exclusive (for writes) and non-exclusive (for reads). So this model has as benefit that a lot of concurrent access is possible, but at the expense that it has gotten a lot more complicated to write programs on such as system.

Finally it should be noted that the consistency models only describes the interleaving of *individual* read/write operations, meaning that to support transactions further concurrency control is usually required, although of course the weak consistency hybrid model described above has this naturally build in.

4 Applying the Theory to the Wind Farm SCADA System

Finally, after all the theory in the previous chapter it is time to analyse how this can actually be applied to the wind farm SCADA system. This is the focus of this chapter. First in chapter 4.1 it is described why it is not possible to directly apply the end-to-end principle by removing all state from the central farm server nodes. Following this, chapter 4.2 analyses how the different availability tactics can be applied to different parts of the SCADA system. Finally chapter 4.3 analyses how the replication theory applies to the data structures and communication used in the system.

4.1 Applying the End-to-End Principle

The easiest solution to the availability problem is to remove all state from the central farm server node, and thereby fulfill the end-to-end principle as described in chapter 3.2. As previously described this will simplify the task of introducing several redundant farm server nodes.

In the following chapters the two most obvious of these solutions for removing network state are described together with the reasons why they are not viable solutions. In summary they are:

End-to-End Notification: In chapter 4.1.1 the idea of having notifications go all the way from turbine nodes to the clients without the multiplexing state in the farm server nodes, is described.

End-to-End Polling: In chapter 4.1.2 the opposite direction is described, with the clients doing full polls all the way through to the turbine nodes, i.e. a situation where the farm server nodes are purely routing the network packets.

4.1.1 End-to-End Notification

The first obvious alternative for removing state from the central farm server, is to let all state with respect to what clients are subscribing to what sensors be stored on the actual endpoints, i.e. the turbine nodes. In theory this should make it possible for the turbine nodes to send the sensor value notification

messages to the clients using any route over a multiple of central farm servers (or directly from the turbine node to the clients without going over a farm server). Unfortunately pushing a direct notification from the turbine node to the client is not possible due to the strict request-response HTTP protocol used by the clients. This direction of the HTTP message flow and the poll-based notifications has already been described in details in chapter 2.2.1. Due to this selection of protocol used by the clients the farm server with its HTTP server is necessary.

Furthermore the turbine nodes should be protected from both intended as well as unintended denial-of-service "attacks" by a larger number of clients and/or rogue clients. This is necessary as the turbine nodes has other assignments besides serving sensor values to clients. Examples of these assignments are to periodically collect data from the rather memory limited RFCs and store it in a local database. Also some alarm surveillance jobs are running on the turbine nodes. And none of these assignments can be allowed to be disrupted or delayed because of a heavy load caused by a larger number of clients. So the multiplexing on the farm server also acts as protection of the limited computing resources on the turbine nodes. With respect to this, it is also worth mentioning that the turbine nodes are low end hardware when compared to the hardware of the central farm server. There are several reasons for this hardware selection, first cost is of course always a factor. Furthermore there is the fact that the turbine nodes are usually placed inside the turbines, i.e. in a rather hostile environment when compared to a normal office or data center. Finally the hardware in the turbine nodes can not always be easily replaced or serviced, as it is often located in rural areas. Therefore the hardware used for the turbine nodes, is, as it is often the case for industrial hardware, selected among the time proven older and stable hardware solutions, which has as effect that some computing power is sacrificed for this.

4.1.2 End-to-End Polling

The first thought after the alternative in the previous chapter has been turned down, is to reverse the picture and let the client polls go all the way through to the involved endpoints, again removing the state from the central server. Besides the resource problem on the turbine nodes mentioned above there are other reasons why this is a bad idea.

First of all letting the client call block until the slowest of the involved turbine nodes has answered is a troublesome road to take, as this will make the client requests "break" or wait on the "weakest link in the chain". Creating a system with such a high coupling and dependency on a large number of nodes for answering a trivial HTTP request does not seem architectural sound. And even in normal operation when everything is running smoothly all client requests will be delayed with at least 2 times the latency[13] between the farm server and the turbine nodes, as this is the fastest theoretical limit for when the turbine node reply can come back if there are no processing time at all on the turbine node.

Furthermore if all state should be removed from the farm server, it is required that all client requests should contain full routing information about values for what sensors on what turbines are required. Otherwise it would not be possible

[13]Or more precise $(4 + 2)$ times the latency when using RMI as described in the detailed RMI analysis performed in appendix F.

for the farm server to route the request to the relevant clients. This must be compared to the situation in the existing system where the client requests for values just consists of sending a single token as all further information is already present at the farm server.

Finally the working threads in the application as it is today are nicely decoupled, separating the concerns of the applications in an attractive way. As an example the interval with which the turbine node schedules the polling of sensors on a RFC are not directly influenced by client requests for values. And the reply speed of the central farm server for the HTTP requests are not blocking on potentially slow RFC reads on a range of remote turbine nodes, so as it is today the farm server can answer immediately with the data it has in its central cache.

4.2 Analysis of Applying the Different Availability Tactics

4.2.1 Failure Semantics

Based on the architectural drivers and QASes it should be clear that the reason for introducing redundancy in the wind farm system, is primarily to increase availability in case of *crash* and *omission* failures for the central node. So in terms of the theory, there are no requirements to handle corruption and byzantine failures.

4.2.2 Availability Tactics

With respect to the availability tactics described in chapter 3.4 it was mentioned that the solution as it is today is an implementation of the *spare* tactics leading to substantial downtime in case of a crash. The arguments for selecting between the other three tactics are as follows:

Voting: This tactic is not relevant due to the fact that only crash and omission failures are to be handled. As explained in chapter 3.4.1 the expensive voting tactic is only the correct tool to bring on board if the system should be resistant to corruption failures.

Active Redundancy: In the situation where the stakeholders are unwilling to accept the tradeoffs of the passive redundancy tactic, and instead strives for continuous high availability of the system, an active redundancy model is the only solution. The architectural challenges for this is discussed in the following chapter 4.2.3.

Passive Redundancy: The pros and cons of the passive redundancy tactic compared to the active tactic, have already been highlighted. When pairing these with the requirements of the wind farm system, where the real-time deadlines for the monitoring of sensor values are only soft deadlines, a passive redundancy model with its simpler implementation seems to be the right solution. The architectural challenges in applying passive redundancy tactics to different parts of the wind farm system is discussed in chapter 4.2.4 and 4.2.5.

4.2.3 An Active Redundancy Tactic

A full active redundancy tactic where all clients and turbine nodes communicate with a group of farm servers, either explicit via multicast messages, or implicit using some middleware layer like DSM is unfortunately hindered due to technical limitations in some of the technologies that are used in the wind farm system. The limitation lies in the communication protocol between the clients and the wind farm server. This is depicted on figure 17, where the *external group* is the part of the system where there are limits to as how the communication between the components can be changed. Opposite to this is the *internal group* that is entirely under the control of the SCADA development group.

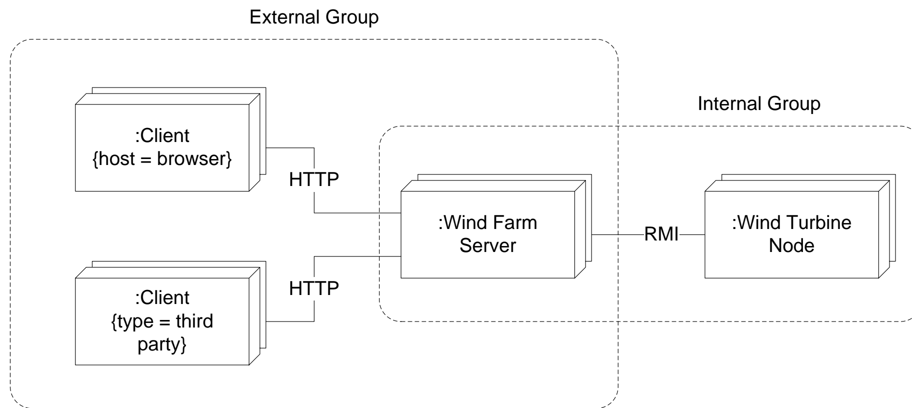


Figure 17: The external and internal group in the wind farm system.

The *external group* from the figure is limited in the following ways:

- The protocol is fixed to HTTP. This can not be changed, both because the standard SCADA client is running in a browser[14] and because an unknown number of third party clients uses this protocol for the existing installations.
- The fact that the protocol is fixed to HTTP, implicitly excludes multicast, and of course also making the client a node in a DSM middleware layer, unless of course the selected DSM software supports node synchronization over HTTP.
- The multiple third party client implementations also has the explicit requirement that the system maintains backward compatibility, meaning that any changes in the protocol must be optional seen from the client side. So using a kind of passive strategy where it is imposed upon the clients to actively change to another wind farm server in case of a crash would be acceptable, as the existing clients will only have to be changed if they need to benefit from higher availability.

[14] To give the full story it should of course be said that the case that the client is a browser DHTML-Ajax based client is not entirely hindering the use of other protocols than HTTP. Using a browser plugin in form of e.g. an ActiveX-component, a Java applet, a Flash component, etc. it would be possible to take advantage of the support in these technologies for other, more feature rich communication protocols than simple request-response. But in the scope of the wind farm SCADA system, this is an area that is not up for discussion.

- Another argument for holding on to the point-to-point HTTP protocol, is that it is relatively easy to inspect HTTP traffic on the fly in firewalls doing traffic analysis, as well as in intrusion detection systems. And although the security of the system is out of the scope of this thesis, it can not be completely ignored on the out-facing side of the wind farm server.

- Finally it should also be noted that the systems often have only limited bandwidth reserved for client communication. Wherefore the communication should not have high ongoing overhead in normal operation (e.g. in terms of multicasts) just to handle the rather seldom error situations.

So to sum up, the communication in the *external group* on the figure, is required to stay as point-to-point communication in terms of HTTP, effectively hindering the introduction of the active redundancy tactic in this part of the system. There is one way around this, referring back to how group communication is often hidden from the clients using *front end* (FE) components, as it was presented on figure 14 on page 30 and in the discussions surrounding the figure. All the above limitations have implicitly assumed that the FE is running in the client process, thereby assuming that the communication between the FE and the farm server group was restricted to point-to-point HTTP.

The obvious solution to this is to move the FE out of the client process and add it as a computing node inside of the wind farm as depicted on figure 18. If the communication on the backside of the FE (the connector marked with a question mark in the figure) would also be HTTP, there are several standard components in form of either a reverse HTTP proxy or a hardware HTTP load balancer that could take this role. But in this exact system, the reason for introducing the FE is explicitly *not* to communicate with the wind farm servers using HTTP on the backside of the FE, so here the FE must be a component that is tailor made to act as a bridge from HTTP on the frontside, to the relevant kind of group communication on the backside, whether in form of multicasts to the wind farm servers, or in form of communication using some middleware layer like for example a DSM layer.

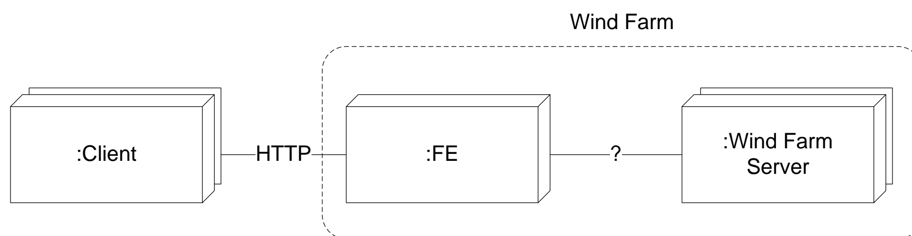


Figure 18: The front end (FE) component placed inside of the wind farm.

Although it seems to be a decent solution, there are a couple of reasons why the FE is not feasible in the wind farm system:

New Single Point of Failure: In the naive implementation of introducing a FE in the system, the new single FE would just be the new single point of failure. I.e. the "chain" has not gotten stronger, the "weakest link" has just been moved from the wind farm server to the FE. The arguments to counter this could of course be that the software complexity of the FE is

simpler than the wind farm server, wherefore it should not be crashing so often due to a smaller number of software bugs. Whether correct or not, this argument does unfortunately not remove the fact that the hardware running the FE will still be subject to crashes to some degree, just as the hardware running the wind farm servers.

Additional Hardware: Introducing a new type of hardware in the wind farm SCADA system is not as simple as it might seem. As it has previously been mentioned in chapter 4.1.1 on page 40 the process of choosing hardware components for the system is a relatively conservative process, favoring stability and time-proven solutions over speed and buzzword compliance. Due to this, it might very well be impossible to introduce a dedicated FE computing unit in the system.

So it seems that the only solution in the communication on the externally facing part of the wind farm server must be a passive solution where the clients only communicates with one active wind farm server, and must switch to a new instance in case of a crash[15].

4.2.4 A Mix of Active and Passive Tactics

Referring back to figure 17 and the different degrees of freedom for respectively the external and the internal group, a solution where a mix of active and passive tactics are used could be another solution. This is depicted on figure 19, where the communication between the clients and the wind farm servers follows the passive redundancy tactic, with communication going to the primary server following the arrow marked with 1 on the figure. Only in case of an error, communication to the backup server along the arrow marked with 2 is taking place.

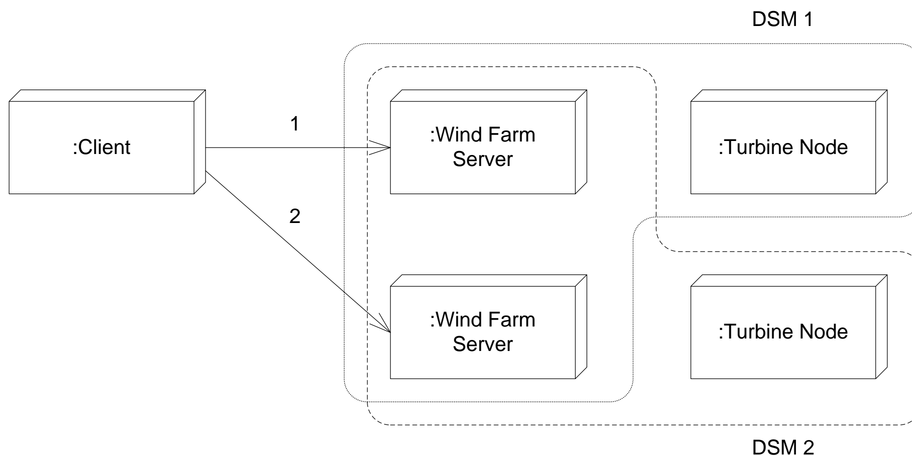


Figure 19: A mix of active and passive tactics.

[15] Although such a run time switch is not without challenges in a DHTML-Ajax browser client, where the so called "same-origin" policy will hinder communication to another server than the one where the JavaScript was loaded from. Fortunately in this case, this security limitation can be worked around using the `document.domain` property as explained in [Mahemoff, 2006, pp.250].

It should be highlighted that there is nothing in this setup that enforces that there is a *global* primary wind farm server, it is only seen from a specific client's perspective that one of the servers are the primary, and the others are backup servers, as it is only the *per client* specific data structures that are passively made redundant. So one clients primary server, could be another clients backup server, which opens for using this setup as a workload balancing solution too. In this case it is of course important to scale the system appropriately to avoid load-related *chain reactions* ([Nygard, 2007, ch.4.2]). In short, a crash due to a load-related chain reaction basically describes the situation where the fact that one node in a workload balanced cluster goes down, of course followed by its work being distributed amongst the remaining nodes (e.g. in a 2-node cluster setup, the workload of the "survivor" node doubles in this case), leads to the other servers being overwhelmed with work and there after crashing due to too much work. I.e. a situation where no benefits are actually accomplished in terms of higher availability.

On the right hand side of the figure, the communication pattern between the turbine nodes and the wind farm servers follows the active redundancy tactic where all communication from a turbine node should be communicated to all wind farm servers concurrently. Conceptually this could be viewed as all the wind farm servers and *one* turbine node sharing one distributed memory region, marked as respectively DSM 1 and 2 in the figure.

It is important to note that since the data structures that must be replicated across the wind farm servers are per turbine node, there is no reason to create one large conceptual DSM spanning *all* turbine nodes, as this would only decrease performance. The relevant data for the replication in this part of the system, is the *sensor values* described in chapter 2.4. In practice all the nodes would probably be part of the same DSM, but with a virtual segmentation of the shared memory pages or objects. The important requirement is that the DSM middleware should make it possible to control that the individual data pages or objects are only replicated to the nodes where they are required.

To analyse the suitability of this solution it is necessary to verify how it handles the replication of the relevant data structures presented in chapter 2.4:

Relations: The relations describing what sensor readings are subscribed to by what clients, are trivially supported by the passive solution on the left hand side of the figure. Due to the above mentioned fact that there are no global primary wind farm server, only the individual turbine nodes have the full picture of the *combined set* of all the sensor subscriptions for a given turbine as shown on figure 20 on the following page. To make it possible for the turbine node to correctly clean up sensor subscriptions that are no longer referenced from any live wind farm servers a traditional lease based protocol between farm server and turbine nodes should be used for sending information about what sensors there are subscriptions for.

Sensor Values: The current sensor values for all currently observed sensors are the data that should be replicated to all the farm servers by the active solution on the right hand side of the figure. A DSM solution seems well suited for this task.

Per Subscription Sensor Values: The values per subscription about whether a given sensor has changed since last polling for changed values by one

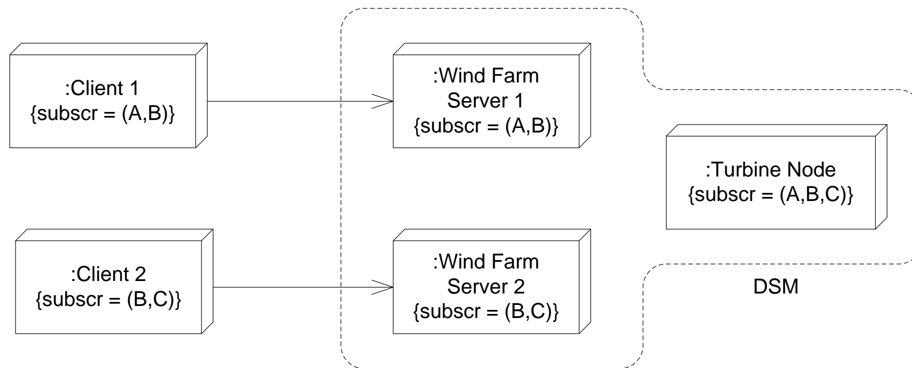


Figure 20: Due to the fact that there is no globally primary wind farm server, only the turbine node has the full picture of exactly what sensors are subscribed to (in the figure labelled (A,B,C)).

client. The passive solution on the left hand side in the system makes it possible to avoid replication of these proxy objects across the farm servers, as these data are only relevant at the wind farm server the specific client uses as primary server. And since these data are the most volatile data in the system this is a big advantage.

The obvious consequence of using the passive solution on these data, is that a client can receive already known data once more if changing farm server. This could be seen as a protocol where a small degree of inconsistency is allowed to get a higher overall performant system.

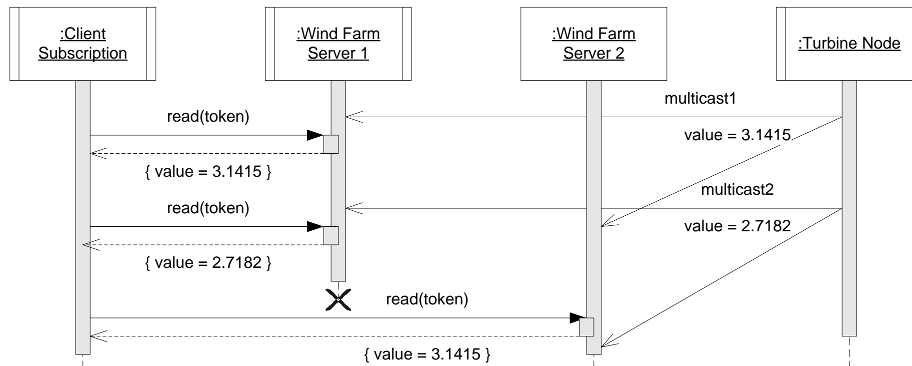


Figure 21: A client changing primary wind farm server, due to a farm server crash. As it is seen this can lead to the client seeing a range of old values for sensors.

Coherence Consistency: As seen on figure 21 the nature of multicast messages also makes it possible that a client will observe a range of old obsolete values when changing from one server to another. A situation that can

not be ignored as just an innocent data inconsistency problem[16]. This is where the theory on memory consistency must be applied. But as this scenario is special in that only one "writing" process (the turbine node) for each sensor exists, there is a trivial solution for this as the data only need to be ordered per turbine node, and not be globally ordered. The simple solution is to let the "writing" turbine node attach a timestamp to each value it sends to the farm servers, thereby in effect applying *coherence consistency* of the data. This solution is basically applying a traditional message sequence number, and whether a simple counter or a full timestamp is used is not vital to the working of the mechanism. But for other usages in the client (such as to be able to visually show age of data) a timestamp might be preferable. Selecting a timestamp of course requires that it is strictly non-decreasing, so no local time with daylight time savings or other jumps backward in time is allowed. So something like UTC time would be preferable.

To sum up the above discussions figure 22 attempts to label the different data structures from figure 13 on page 22 with information on what kind of redundancy tactics are used on each structure.

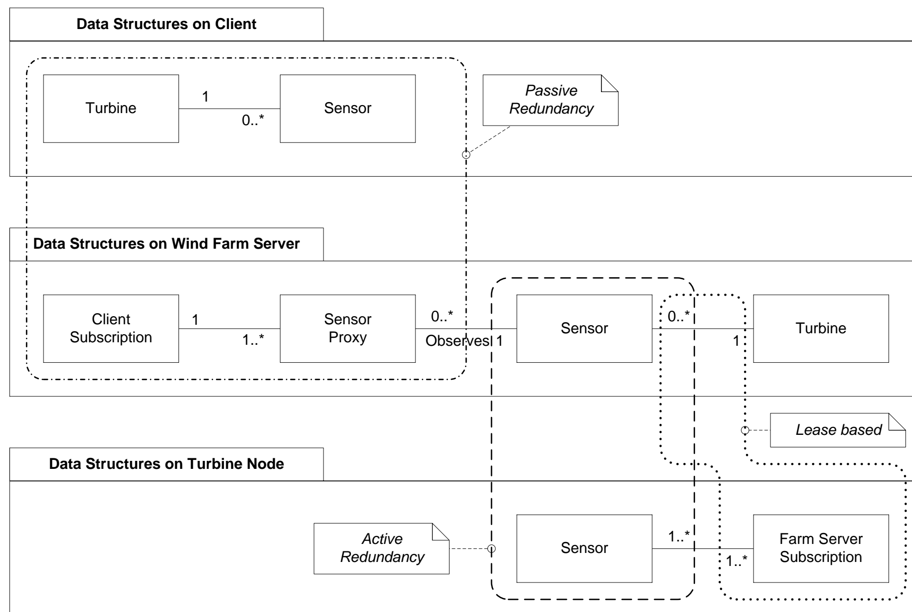


Figure 22: The different types of redundancy and what data and relations they span over, across clients, wind farm servers and turbine nodes.

4.2.5 A Passive Redundancy Tactic

The final proposal for a solution would be to use the passive redundancy tactic on the entire system, i.e. also for the communication between the wind farm

[16]The opposite scenario, where a client misses some values on a server change is of course also possible. But since the values represent *states* and not *operations on state* such missed values are not a problem.

servers and the turbine nodes. Thinking about such a solution, there does not seem to be any obvious problems as to why this should not be able to fulfill QAS1 - 3 as the 5 seconds failover time in QAS1 should be plenty of time to *detect* a failing farm server, *elect* a new active server and *propagate state* to the new active server. Furthermore such a solution would probably be the least intrusive with respect to the existing system, which might make it the most cost efficient in terms of implementation effort and complexity.

Although the above sentences might give the impression that a passive solution is entirely trivial to implement, this is not the case. There are still quite a few architectural challenges that will have to be considered to implement a solid solution. These challenges are described below.

Crash Detection: The first problem is for the passive node to detect when the primary node has crashed. For this one of the fault detection availability tactics from [Bass et al., 2003], like *ping/echo* or *heartbeat* would be obvious candidates for solutions. If network partitions inside of a wind farm must also be handled this crash detection suddenly becomes a hole lot harder. A part of the crash detection problem, is also the problem of *election* of the new active node, in case of a system where more than one passive node should act as takeover system.

State Propagation: When the passive redundancy tactic was described in chapter 3.4 it was mentioned that in a system with passive redundancy it should be decided how much shared state that should continuously be synchronized between the active node and the passive node(s), and how much state that should lazily be loaded from for example shared persistent storage on the time of takeover. In the wind farm system with its relatively simple data structures at the wind farm server, the best solution seems to be, to let the connected clients all resubmit their subscription requests at a takeover situation, meaning that no continuous synchronization between active and passive node(s) has to happen, with respect to the subscription states[17]. It is of course obvious that the requirement that the clients should be able to resubmit all subscription calls, puts an additional burden on the logic in these. Furthermore it means that the handling of the failure of a crashed active node, is propagated to elements in the system that would perhaps benefit by being shielded from this. So whether it is wisely to make the decision to include the clients in handling the takeovers is not entirely clear. Another argument against this decision, is that the clients are not always created by the manufacturer of the SCADA system, as some of the clients are third-party integrations. Meaning that it is harder to verify that all the possible clients actually behaves according to the rules, and will be able to handle the takeover situations correctly.

[17]There is other state that has to be continuously synchronized though. Most obviously the information on active sessions, and the information related to those such as authentication, authorization and session token information. But this is outside of the scope of this thesis.

4.3 Applying the Replication Theory

4.3.1 The Types of Replication Messages

With respect to the distinction between *replicating state* or *operations on state* as described in chapter 3.5.1 it is easy to see that for the wind farm SCADA system the messages sent from turbine nodes to the farm server contains the current state, i.e. current values for a list of sensors. With respect to the subscription modification messages sent from the clients to the farm server the opposite is the case, as these are actually operations on the overall subscription states in the farm server.

4.3.2 Failure Semantics for the Group Communication

The group communication required especially in the active redundancy tactic makes it necessary to consider how the theoretical areas described in chapter 3.5.2 and 3.5.3 applies to the system.

Group Size: As only crash and omission failures should be handled, and as the system needs to be tolerant to 2 concurrently failing farm server nodes, the group size of the farm server nodes is 3.

Group Membership: The physical layout of a wind farm is very static, leading to the observation that *static* group membership is sufficient for the SCADA system. Furthermore as there are no transactions in the system there is no need for groups views or other complex group concepts.

Multicast Reliability: Since the *internal group* of the system where multicast of messages might be used is a LAN, it will make sense to consider if raw IP-multicast communication is suitable, as LANs usually gives a high delivery guarantee, even for IP-multicast messages.

Furthermore as the architecture does not handle the scenario where the network is segmented in a way where a client can only connect to a farm server node that is in a network segment where the communication to the turbine nodes is down, it probably does not make sense to use an expensive reliable multicast protocol between the farm server nodes and the turbine nodes.

For each of the individual practical solutions described in chapter 6 these areas will be elaborated upon when there are detailed information about these.

4.3.3 Memory Consistency Models

As the *sensor values* are the only replicated data when an active redundancy tactic is used in the *internal group* (as seen on figure 22 on page 48), it is only relevant to consider the memory consistency model for this set of data. For these data it should be observed that there is only one “writing” process per memory location, as one turbine node is responsible for collecting data for its own sensors, where the current value of one sensor can be seen as one memory location.

This property together with the business fact that there is no requirement for globally ordered data, but only a requirement that the data from each individual turbine must be strictly ordered, makes it easy to accomplish *coherence consistency* by applying the message sequence number mechanism described on page 47.

If the solution is implemented using a DSM instead of explicit multicast messaging this means that a *hybrid* per *entry consistency* mechanism will be sufficient, as this granularity leaves room for optimizations at the higher levels. This will let the higher levels communicate knowledge of batching updates together in one flush write operation to the DSM system.

5 Solution for the External Group

As described in chapter 4.2 and shown on figure 19 on page 45 a simple passive redundancy tactic must be used for the *external group*. In this chapter it is described how this is actually implemented - an implementation that is shared by all 3 prototypes described in the following chapter 6.

First in chapter 5.1 it is described how a client node detects that the currently active farm server node it is using has crashed. After that, in chapter 5.2, the recovery procedure with electing a new active farm server and transferring relevant state to that, is described.

5.1 Failure Detection

In the external group a client must detect if the wind farm server node it is using as the active server is crashed. Due to the poll-based mechanism for event callbacks described on page 16 this is straight forward, as a missing response to a read-call will let the client node deem the wind farm server node as being crashed. This mechanism is shown on figure 23.

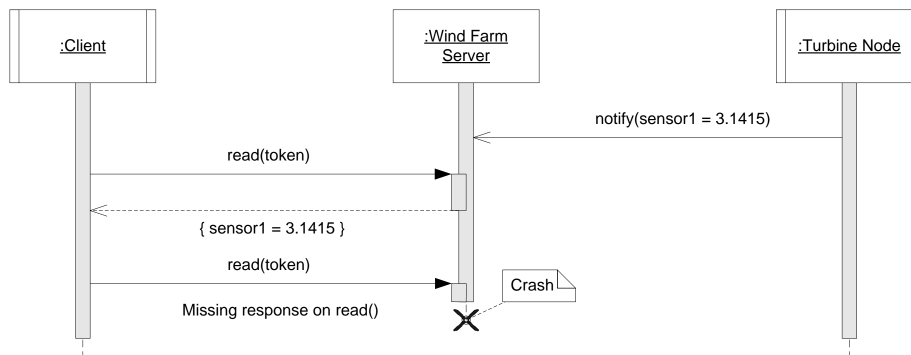


Figure 23: Client failure detection of a crashing wind farm server node.

5.2 Recovery

When a client detects that its active wind farm server node is crashed the recovery process must be initiated. In the external group this happens in two steps shown in figure 24 on the next page:

Election: Selecting a new active wind farm server node is done by trying to connect to all 3 farm server nodes in parallel. The server that answers first wins the election and is selected as the new active server for the client. This simple algorithm has the side effect that the work load when a high number of clients are present, will be distributed across the alive wind farm server nodes, as lightly loaded nodes will typically answer faster than heavily loaded nodes.

State Transfer: For the external group the only state that must be transferred to the new active wind farm server is the information on what sensor values the clients using this server are subscribed to, i.e. the so called *relations* as described in chapter 4.2.4. This is easily done by letting the client resend it's active subscription information to the newly selected server.

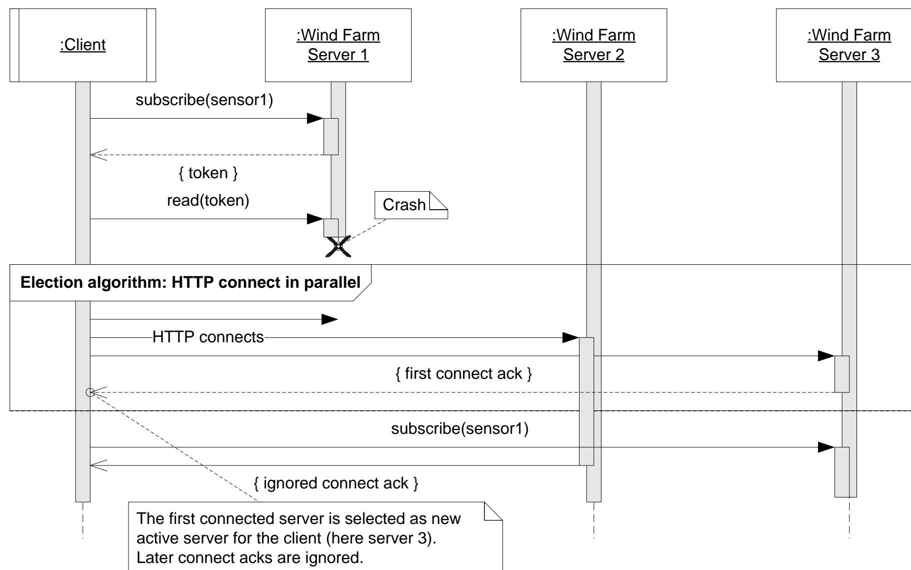


Figure 24: Election algorithm for selecting the new active wind farm server node for a client.

With respect to the election algorithm it should of course be mentioned that the only reason why it is possible to do the election in parallel and just ignore the servers answering “late”, is because the HTTP connect calls does not change any state on the wind farm servers.

6 Solutions for the Internal Group

To implement the different possible redundancy tactics for the internal group described in chapter 4.2 several different solutions exists in the Java space. This chapter documents 3 prototypes implementing these tactics in different ways.

To view these solutions from a higher perspective this chapter sets of with a short description on how it is possible to add the new availability qualities to the existing system using either transparent wrappers or explicitly using toolkits

(chapter 6.1). This leads to two different prototypes implemented with either of these two solutions:

- A solution with a DSM region per turbine using a *passive* redundancy software product [Terracotta] described in chapter 6.2. This started of as a transparent solution, but due to lacking performance it had to be changed to an explicit solution using a Terracotta specific API.
- An explicit solution using network multicast using the software product [Hazelcast] described in chapter 6.3. This leads to an *active* redundancy solution.

Thereafter in chapter 6.4 a prototype based upon the end-to-end principle with "soft state" is implemented by manually extending the existing system using standard Java Development Kit (JDK) network APIs.

To follow the discussion of these 3 prototypes it is an advantage to be at least somewhat familiar with the Java platform and the central APIs in the JDK. The source code for the 3 prototypes are available in the source archive for this thesis.

Finally a number of other possible technical solutions are shortly described in chapter 6.5.

The actual evaluation of how each of the 3 prototypes fulfills QAS1 - 3 is done in chapter 8 using the test bench described in chapter 7.

6.1 Transparency of the Solutions

[Birman, 2005, ch.20] describes retrofitting reliability onto existing systems. As part of this he touches on how different kind of solutions requires different degrees of modification of the existing system. Basically he distinguishes between two groups of solutions:

Wrappers: The ideal goal of a wrapping solution is to transparently *wrap* the existing software and thereby applying the target availability qualities without having to modify the existing software at all. This ideal is similar to the transparent wrapping that takes place several places in the TCP/IP network stack, for example a thing such as encryption in form of TLS/SSL can be applied transparently, typically without having to modify the application using the network stack. Birman describes several different wrapping techniques, that are not necessarily all equally appropriate given the technology used in the wind farm system:

- **Wrapping at object interfaces:** At first glance this wrapping technique seems adequate in a object oriented Java world. And it would surely be if the existing system had been implemented in a way where all communication has been hidden away from the application logic, e.g. in form of Java enterprise beans where the JEE container typically makes it entirely transparent for the individual beans whether they are communicating with other beans as local or remote objects. In the existing wind farm system, the communication is already made explicit though, so solutions that uses wrapping at object interfaces does not seem as obvious candidates.

- **Wrapping by library replacement:** This is typically a technique used in systems build in traditionally linked programming languages such as C, C++, Pascal, etc. Basically it describes the concept of changing the overall system properties by changing which libraries the system are linked against (whether dynamically or statically linked). This technique is not relevant to apply in the Java world[18].
- **Wrapping by object code editing:** This technique is even more intrusive than wrapping by library replacement, as this requires modifying the programs object code. In programs compiled for a specific target OS and platform, this object code is typically represented in form of processor specific machine instructions. The same applies to programs running in a VM, where the machine instructions are instead just VM specific (e.g. in Java, in form of Java byte code, and in the Microsoft .NET platform in form of Common Intermediate Language (CIL)).

Toolkits: Where wrappers in their purest form does not require modifying existing code, the toolkit approach is entirely opposite, as this requires rewriting the application to explicitly take advantage of the toolkit by using the API and protocols it offers.

It should be clear that these two types are at opposite ends of the spectrum of possible solutions, and that different kinds of mixed solutions also exists. Since the wind farm system, as mentioned already uses explicit remote communication, it would of course require modification of the code in the system to use a wrapper solution, in which case it is of course no longer a pure wrapping solution.

A word of caution on the wrapping solutions seems appropriate though, since as it is always the case when the distributed mechanisms in a system are completely hidden and transparent to the programmers one should be a little bit skeptical as described in both the classical critique of transparency in [Waldo et al., 1994]'s *Note on Distributed Computing* as well as in less academic terms in *the law of leaky abstractions* ([Spolsky, 2004, essay 26]). It might very well be easy to create a distributed application without detailed knowledge of the underlying details, but if done naively it might fail and burn in flames at the worst possible time when put under heavy load in the production environment, or have unexpected failure scenarios when the underlying network details leak through the abstractions in unexpected ways in case of network errors. Of course this should not be seen as a praise entirely against abstractions and frameworks striving for full transparency, but more as a word of warning, that even though it might *look* easy, it is still required that the development team has an intimate knowledge of what is actually going on in the internals of a given framework.

Along the same skeptical line is Birmans words on the fact that wrapping solutions traditionally are more limited than explicit toolkit solutions. But of course as always the cost of choosing a toolkit over a wrapping solution should be weighted.

[18]Although this is actually what happens at JVM level when one changes between the so called client and server VM (with the java -server or -client options). This changes the implementation and behaviour of the VM by using two different dynamic link libraries.

6.2 A Passive Redundancy DSM-Based Solution Using Terracotta

The first prototype is a DSM-based solution that is build using [Terracotta]. The source code for this prototype is available in the source archive for this thesis in the source/terracotta folder. Besides the 4 standard project modules described in appendix C (client, server, node and common) the folder contains the Terracotta 3.5.1 release used in the prototype.

The Terracotta project is a platform covering a suite of solutions in the area of distributed objects on the Java platform. Of these solutions a distributed cache product, called Ehcache is the main product. But in the context of this thesis the Terracotta project is mostly interesting because of the underlying technology upon which their entire solution suite is build. A technology coined "network attached memory" (NAM) depicted in figure 25. In [Terracotta, 2008] this is described in overall terms as:

> *At its core, Terracotta's goal is to allow several computers to communicate with each other as if they were one big computer through memory and pure memory constructs, such as threads and locks (mutex objects and the like).*

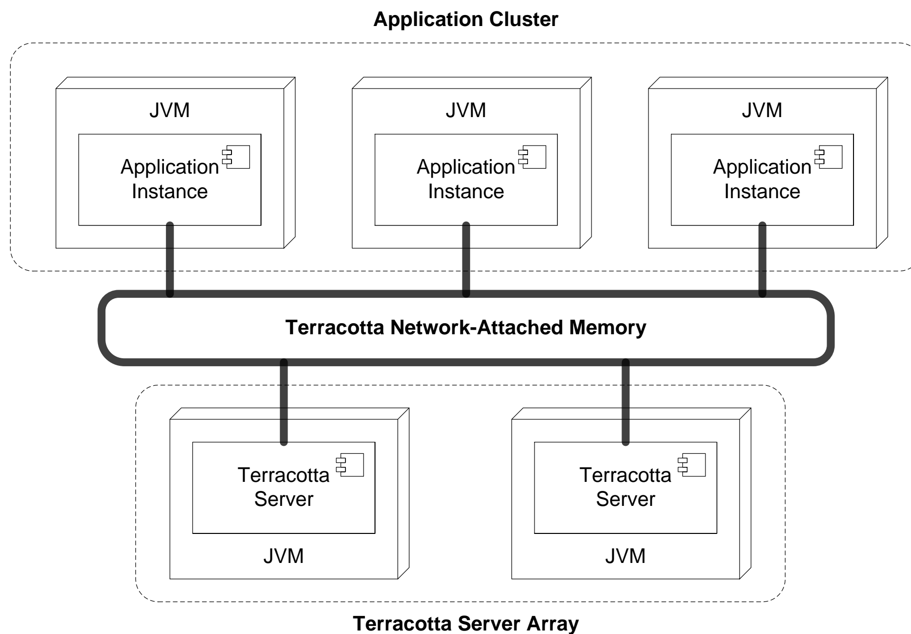


Figure 25: Terracotta Network Attached Memory (NAM).

As it should be clearly visible from the figure this is actually a DSM system, although coined with their own fancy term. The innovative and exiting about Terracotta NAM is their implementation of the memory consistency model, which is described in the sections below together with the other technical properties of the Terracotta platform.

Passive Redundancy: Terracotta is a classical implementation of a passive

redundancy solution working with an active node, and one or more passive nodes.

Memory Structure: A DSM has to make some kind of "memory chunks" available to the processes using the DSM. Traditionally the different DSM implementations has fallen into one of three groups:

- "byte oriented" making the raw memory bytes available to the programs. This is of course mostly suitable in low level programming languages like C.
- "object oriented" making the "chunks" of memory in the DSM visible as some kind of objects to the programs.
- "tuple oriented" where an explicit API are to be used by the programs using the DSM. This will consist of atomic operations like "write" (add a tuple to the space), "read" (read a tuple), "take" (read and remove a tuple). The explicitness of the tuple oriented solutions are in contrast with the other two solutions where it might not be explicitly visible for the programs whether they are working on a byte structure or object that are part of the DSM or whether it is traditional local memory.

In Terracotta the object oriented concept is used, where the shared objects are called "distributed shared objects" (DSO). Seen from the Java programmers view the DSOs are not in any way different from traditional Java objects.

The decision of which objects should be distributed in the NAM is made by configuration of Terracotta where one or more objects are configured as so called "root"-nodes, and any object *reachable* from a root node are distributed and part of the NAM. Where the concept of *reachable* is the well known concept used in many other computer science areas, such as tracing garbage collector algorithms, and serialization algorithms such as the standard Java Serialization.

Memory Consistency Model: The memory consistency model used in Terracotta is a hybrid model that falls into the category of *entry consistency* models. As the entry consistency model leaves it to the programmer to control the exact fence and lock operations this seems to be contrary to the previous claims about the solution being a transparent wrapper. This is solved in Terracotta by extending the standard Java synchronization mechanisms so that these in a Terracotta solution no longer is used only for local JVM synchronization between memory reads and writes from different threads, but instead is used for DSM wide memory synchronization.

Implementation: Terracotta is clearly a wrapping solution, where the transparency to the programmer is implemented by runtime bytecode injection of the byte code for the DSO objects. Runtime bytecode injection works by modifying the Java byte code for the relevant classes as part of classloading, whereby it in the Birman taxonomy places Terracotta as a wrapper implementation in form of *wrapping by object code editing.* In the later versions of Terracotta the bytecode instrumentation solution seems

to be deprecated in favor of standard Java Serialization as this should offer better performance.

State Transfer: The transferring of memory state between nodes in the DSM in Terracotta can be fine tuned to use either transferring of *full object state*, or to record *operations on state*, and transfer these operations to the different DSM nodes and apply the operations locally. In the prototype build here transferring of full object state has been used.

6.2.1 Implementation Notes

Since the original code used explicit network callbacks to communicate sensor values from the turbine nodes to the farm server, this part of the application had to be changed to use a standard Java ConcurrentMap instead, and then let Terracotta handle the job of distributing the values in this map transparently to all relevant nodes. This change is depicted on figure 26 which is a copy of the original figure 5 on page 14 with the changed parts marked in red.

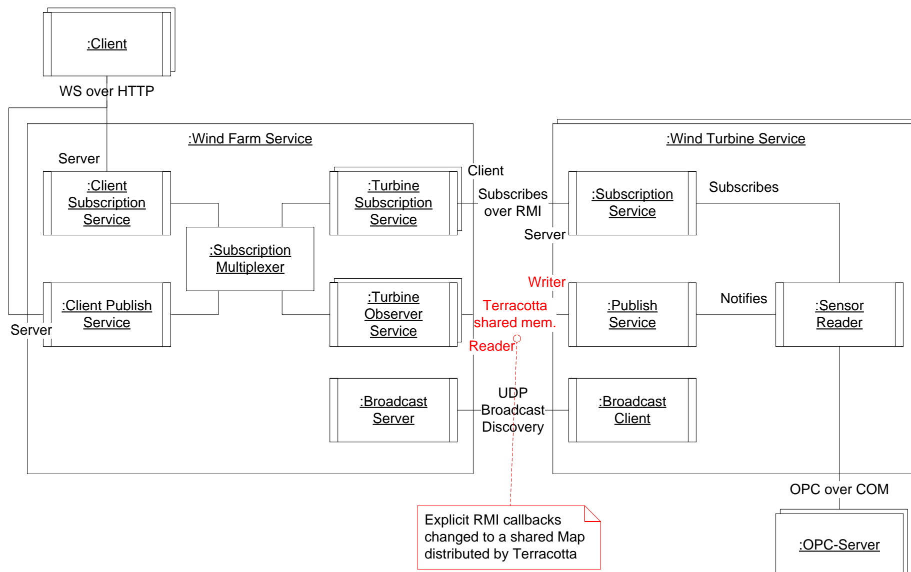


Figure 26: Component and connector diagram for the Terracotta based prototype.

It should be noted that a map instance exists for each turbine node leading to the DSM being segmented into several regions as depicted on figure 27 on the following page.

The prototype started out by being a completely transparent wrapping solution by using the Terracotta configuration files to mark the map instances that should be part of the DSM as "root"-nodes in Terracotta vocabulary. Although this seemed to work fine in small scale with 3 nodes (one of each type), the performance when put on the test bench described in chapter 7 was lacking. After trying to resolve this purely by optimizing the Terracotta configuration it was therefore attempted to change the prototype to use the explicit Terracotta Toolkit API described in [Terracotta, 2011, ch.7]. Basically this gives the

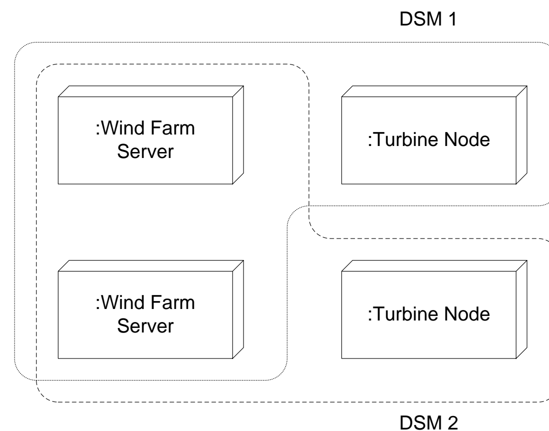


Figure 27: The DSM is segmented into one region per turbine node.

source code access to use some Terracotta optimized concurrent collections as the "root"-nodes.

6.2.2 Analysis of Pros and Cons

As described in detail in chapter 8 the attempts to get the Terracotta based prototype to fulfill any of the QASes failed. This in itself of course disqualifies the prototype, but the pros and cons of it are listed below anyway.

- The usage of the standard Java synchronization primitives for the memory consistency is a clever solution making it a good fit for programmers used to concurrent programming in Java as it follows the same line of thought as the standard memory model in the JVM as described in [JSR-133, 2004] and [Goetz et al., 2006].
- The Terracotta documentation describes only running the Terracotta servers in separate JVM instances. This in itself is not optimal for the wind farm SCADA system, as it either requires new dedicated hardware nodes or that the wind farm server nodes runs two JVM instances, one with the SCADA program and one with a Terracotta server. After a bit of experimentation it was possible to host the Terracotta server as a component in the wind farm server instance though. It should be mentioned that integration with the startup and shutdown of the embedded Terracotta server is a bit rough.
- The Terracotta platform is quite big, which of course could be interpreted as both an advantage in that it can probably solve a multitude of problems if one is very familiar with the configuration options and its inner working. On the other side it is a disadvantage if one wants to use only a specific functionality of it in an existing system as there is a big payload both in form of concepts and documentation (of which quite a lot is not relevant to the usage here) and in form of a big binary payload, where the jar file containing the wind farm server node went from approximately 1 MB to 33 MB and the startup time of a server node went from 1-2 seconds to 20 seconds.

- As described on page 54 there are certain risks in using a completely transparent wrapping layer or toolkit for something that is inherently hard, and that is exactly what was experienced when working with the Terracotta system. It was seen that the performance was not as expected, neither in the normal situation with everything running stable, nor in the situations with crashes, but although extensive debugging and manual reading was done, it was not possible to find out exactly what happened and went wrong. The only thing that could be observed was extensive network traffic (which as described in details in chapter 8 was orders of magnitude larger than for the other prototypes) and measurements showing that the memory consistency was not as expected.

As a concluding remark it should be said that although it was a failure to get Terracotta to perform in the prototype, this should probably more be attributed to missing competences of the author of this thesis, than to Terracotta itself, as it has otherwise been given impressive reviews in the press.

6.3 An Explicit Active Redundancy Solution Using Hazelcast

The second prototype is an active redundancy solution build using the multicast features of [Hazelcast]. The source for this prototype is available in the source archive for this thesis in the source/hazelcast folder. Besides the 4 standard project modules described in appendix C (client, server, node and common) the folder contains the Hazelcast 1.9.3 release used in the prototype.

Hazelcast is a Java library with APIs making it easier to build clustering and distributed network communication into applications. It contains a multitude of features that can be used individually. A few of these features are listed below in the words of the [Hazelcast, 2011] documentation:

- Distributed implementations of the standard Java collections: java.util.{Queue, Set, List, Map}.
- Distributed Topic for publish/subscribe messaging.
- Support for cluster info and membership events.
- Dynamic discovery.
- Dynamic failover.

The technical properties of the Hazelcast library is described in broad in the following sections, although only the multicast functionality has been used in the prototype:

Active Redundancy: Hazelcast is a pure active redundancy solution working with no dedicated master node. For the distributed java collection implementations the data is distributed evenly across the nodes, with a backup placed on one other node. For the map-type this for instance means that all calls to access a range of the keys for the map hits one server “owning” this range of the keys.

Memory Consistency: For the distributed collections it makes sense to consider memory consistency. The default settings in Hazelcast gives a very strong *sequential consistency* guarantee for each individual collection, implemented using the mechanism with each server owning a part of a collection as described above.

Multicast: The com.hazelcast.core.Topic class makes it possible to send ordered multicasts. Based on the documentation it is unclear exactly what ordering guarantees are given by the implementation, but for the usage in the wind farm SCADA system it has been trivial to manually ensure the required FIFO ordering on top of the Hazelcast multicast by letting each turbine node include a timestamp in the sent messages. A solution that of course only guarantees ordering because there is only one "writing" turbine node for each sensor. For the reader familiar with the Java Message Service (JMS) API it should be noted that the Hazelcast Topic is not an implementation of the JMS Topic-class, although both has the same overall usage scenario of publishing a message to several subscribers.

6.3.1 Implementation Notes

Since the original code used explicit network callbacks to communicate sensor values from the turbine nodes to the farm server, it was trivial to change the sending end (the *publish service*) to publish callback messages to a Hazelcast Topic and then let the Hazelcast library take care of distributing the callbacks to all alive wind farm servers. In the wind farm server end, it was also simple to change the *turbine observer service* from being marked as a Java RMI remote callable class to implement the Hazelcast MessageListener interface, and register the instance for messages on a the topic. One global topic for all messages from any turbine node is used in the prototype.

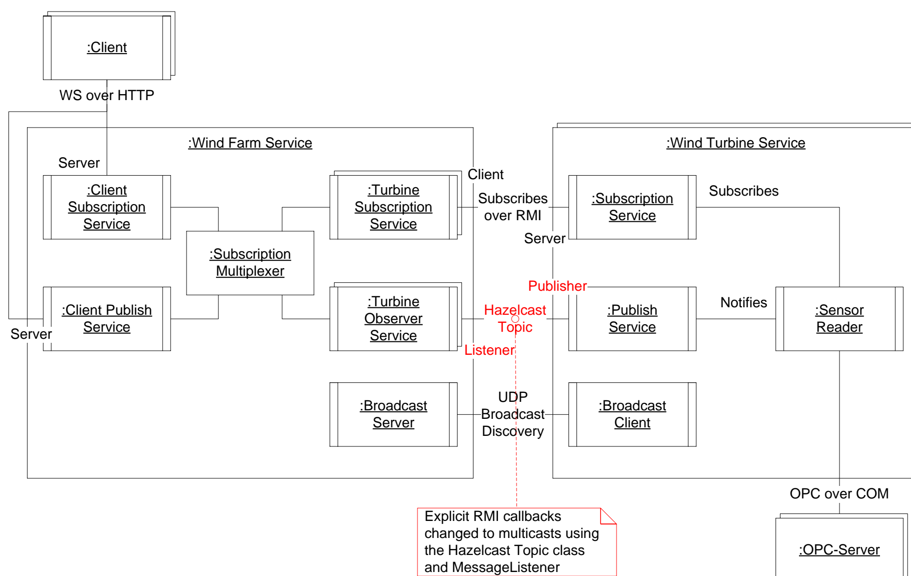


Figure 28: Component and connector diagram for the Hazelcast based prototype.

The changes described above is depicted on figure 28 which is a copy of the original figure 5 on page 14 with the changed parts marked in red.

Although the changes in figure 28 seems similar to the changes done in the Terracotta based prototype, it should be stressed that it was a lot simpler to implement the Hazelcast prototype, primarily due to the fact that the solution is a straight forward evolution of the original unicast solution to a multicast solution.

6.3.2 Analysis of Pros and Cons

As it will be described in chapter 8 the Hazelcast based prototype in general is close to fulfilling the QASes, although there is at least one bug in either the Hazelcast library itself or in the usage of the library in the prototype that will need to be found and removed before the prototype will be usable in the real solution. The pros and cons of the Hazelcast prototype are given below.

- Hazelcast is a pure library, and not a platform in itself. For existing systems like the wind farm SCADA system, this makes it a lot simpler to embed, and pick just the requested features.
- The fact that Hazelcast is an explicit toolkit and not a transparent wrapper of course increases the coupling of the system to Hazelcast. But for all of the distributed collections offered by Hazelcast it should be said that these of course implements the standard interfaces found in the Java Collections Framework, although this is not the fact for the multicast functionality used in the prototype.
- The multicast feature in Hazelcast makes the prototype really simple, when compared to all the manual state-keeping and manual asynchronous callbacks that had to be implemented in the hand coded prototype described in the next chapter. Seen from the programmers view a lot of the "heavy manual lifting" is removed by using the Hazelcast library.
- The payload of the used features of the Hazelcast library are minimal when compared to the Terracotta platform. This goes both for the binary payload as well as for the server startup time.
- The documentation of the Hazelcast library is well balanced between information that focuses on how to use Hazelcast, and information about the lower level technical details about how it is actually implemented. It should be said though that for the multicast features used in the prototype, the documentation on the low level technical details, for instance with regards to the ordering guarantees, is missing.

To summarize it can be said that it was all in all a pleasure to implement the Hazelcast based prototype, and at first it was also considered to be on par with the hand coded prototype described in the following chapter, but as it will be described in chapter 8 a couple of often occurring bugs that makes it fail in fulfilling the QASes, was found.

6.4 An End-to-End Solution with Soft State

The third prototype is a handcrafted solution that is neither based on active or passive redundancy of the central wind farm server state, but instead is build upon the *end-to-end* principle described in chapter 3.2 and chapter 4.1.

The prototype is build without the help from any third party libraries, and build entirely with the standard JDK network and thread APIs. The source for the prototype is available in the source archive for this thesis in the source/end2end folder.

6.4.1 Conceptual Description of the Solution

As described in chapter 4.1 the obvious end-to-end based solutions with no state at all on the wind farm server nodes are unsuitable for the SCADA system. But as described in [RFC 3724] it is possible to build a solution fulfilling the end-to-end principle by storing only recomputable state, or so called “soft state” at nodes between the end-point nodes.

This is the exact idea behind the solution in the third prototype, making it possible to view the redundancy in the solution as a form of “redundancy in network routes”, i.e. if a wind farm server crashes, the individual clients should independently determine a new route over another wind farm server, to the end turbine nodes.

The main algorithm is best described as:

1. Let all turbine nodes register to all live farm server nodes. Seen from the turbine nodes, the farm server nodes should just be seen as independent clients subscribing for sensor value callbacks.

2. In case a turbine node detects that a farm server node is crashed, all state related to that farm server node should be thrown away from the turbine node.

3. The passive redundancy tactic used in the external group (described in chapter 5) should then automatically move the crashed subscriptions to one or more live wind farm servers, without involving any explicit work on the turbine nodes, or without any state having to be replicated between the wind farm server nodes.

A sequence diagram showing an example of this, where two clients subscribes to a set of 3 sensor values from the same turbine node is shown on figure 29 on the following page. The diagram shows how the two clients starts by using the same wind farm server (no. 1). After this wind farm server crashes, the two clients independently selects two different wind farm server nodes for re-establishing the communication to the turbine node (shown as the red circle numbered 1).

The diagram also depicts the changes in the callback state stored at the turbine node. At the time where the turbine node detects the crash of wind farm server 1, the state for this server is thrown away on the turbine node (marked with the red circle numbered 2). Finally the red circle numbered 3 describes the re-established state at the turbine node after the clients have re-subscribed using respectively server 2 and 3.

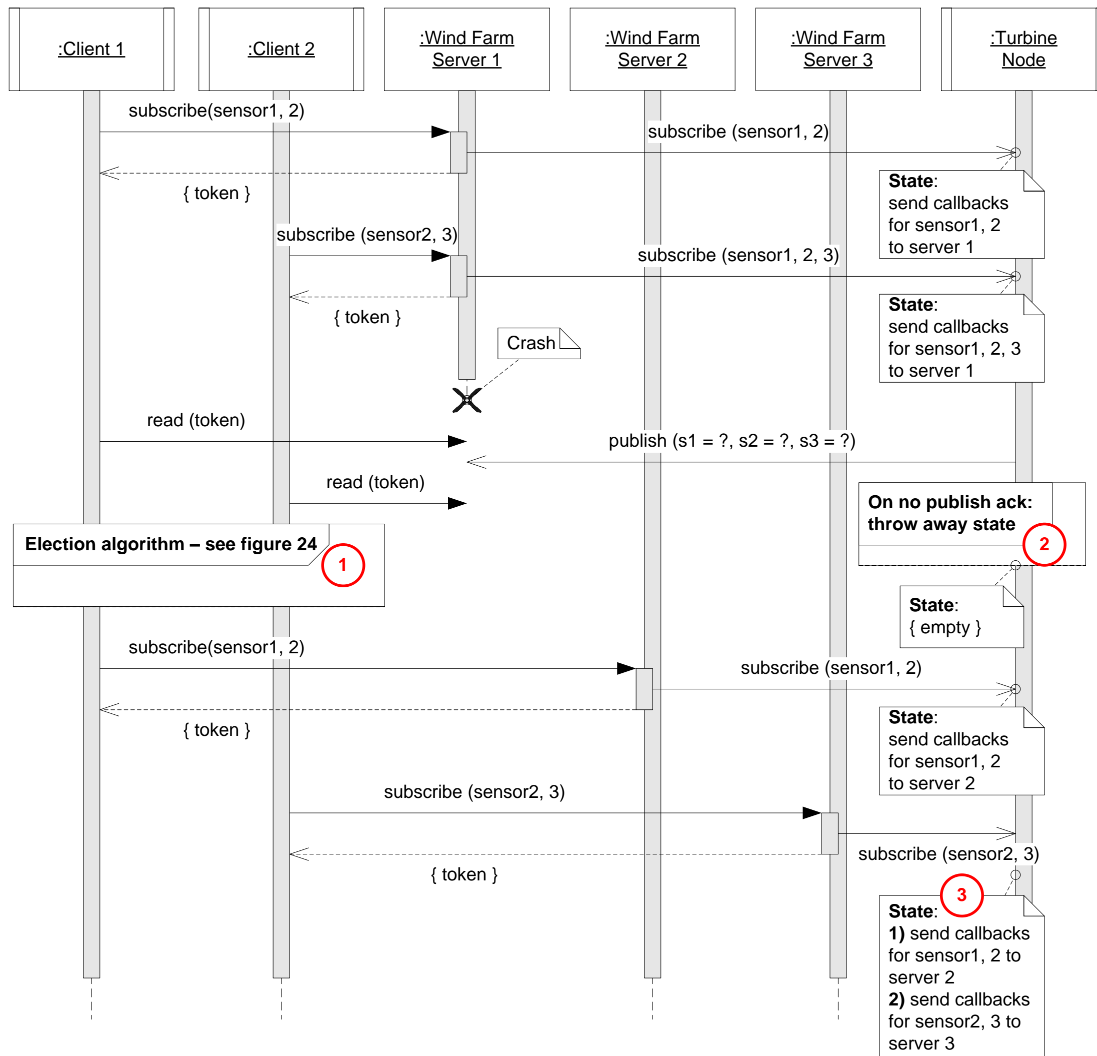


Figure 29: Sequence diagram for the main mechanism in the handcrafted prototype.

6.4.2 Implementation Notes

Extending the wind turbine node source code to explicitly keep track of the subscriptions from a number of wind farm servers is a natural evolution of the original solution where a single wind farm server node exists. The main challenging part of the solution was to keep track of the parallel non-blocking communication that it was necessary to introduce. This parallel communication is of course necessary as a single non- or slow-responding wind farm server would otherwise degrade the performance of the entire system. As the RMI communication used between the wind farm server and the turbine nodes does not allow for asynchronous non-blocking communication it was necessary to build this communication pattern manually using standard multi-threaded programming. In the source code for the prototype this manual multi-threaded code is particularly

present in the turbine module in the dk.accel.misw.mp.model.node.impl package, that is littered with usages of Executors from the JDK java.util.concurrent package.

The crash detection, where a turbine node detects that a wind farm server has crashed is easily implemented as this is signaled clearly by the standard RMI mechanism. Upon detecting such a crash the turbine node dismisses all state about relevant sensor subscriptions previously made by the crashed server, and waits for one or more other wind farm servers to establish these subscriptions as the clients re-establishes the communication through these wind farm server nodes. Separately from this the turbine node repeatedly attempts to connect to the crashed server again, to signal to it that its subscriptions has been dismissed.

This last step where the turbine node actively attempts to signal to the crashed farm server is important, since it would otherwise be possible to see a situation where a temporary communication disruption between a wind farm server and a turbine node, would lead to the turbine node dropping the callbacks, and the farm server just waiting indefinitely for the callbacks.

6.4.3 Analysis of Pros and Cons

As it will be described in chapter 8 this prototype is successful in fulfilling all the QASes, but is does have it negative sides anyway. So below follows the pros and cons of this prototype.

- The solution can be seen as simple, as there is no multicast operations involved and no active or passive redundancy. But at the same time the solution can be seen as complex, as the success of the availability of the solution highly depends on all 3 node types “playing by the rules”, as they are all involved in re-establishing the overall correct system state when a wind farm server node crashes.

- From the turbine nodes perspective the system is really just a network of clients, as the multiplexing of sensor subscriptions on the wind farm server nodes, makes the end client nodes “invisible” to the turbine nodes. Furthermore it simplifies the logic on the turbine nodes that it should just throw away the subscription state related to any crashed wind farm server node.

- The implementation of the communication between the wind farm server and turbine nodes is tedious and requires a high degree of attention to the small details, to juggle the multi-threaded programming related to the asynchronous communication, to make sure that a crashing node does not block communication to other live nodes. So although the prototype shows it to be possible, there is a risk that a solution based upon this code will be a hassle to maintain and evolve over time, as there are quite a lot of small details in the communication to keep in mind.

- As the prototype is created with a very specific problem in mind, there is absolutely no handling of other plausible network problems, such as for instance network segmentation. This does not say that this is necessarily handled gracefully in the two other prototypes using third party code, but there is at least a bigger chance that some of the failure scenarios are actually handled in a well defined way out of the box by these solutions.

6.5 Solutions Not Evaluated

A number of other technical solutions could have been selected for building the prototypes. The most obvious of these are listed below:

Gossip Architecture: The Gossip architecture is described in overall terms in [Coulouris et al., 2005, ch.15.4]. It is an implementation of the active redundancy tactic with the update propagation between the redundant nodes taking place lazily in form of so called "gossip messages". Coulouris et al. points out that the architecture is inappropriate for updating replicas in near real-time due to the lazy communication, and that a solution based upon multicasts will be more appropriate. This is exactly what the Hazelcast based prototype is.

Distributed Event System: Instead of manually building the end-to-end solution with soft state it might have been worth looking at a framework for building a distributed event system as described in [Coulouris et al., 2005, ch.5.4]. This would have automatically lifted the task of asynchronous communication that was hand coded in the end-to-end based prototype. In the Java world it would be obvious to look into an implementation of the [Jini] specifications for such a system.

Message Bus Architecture: [Birman, 2005, ch.11.6] describes an architecture called the "message bus architecture" that can be used for communication in an active redundancy architecture. The superficial inspection that was performed of this, indicates that it is similar to the JMS Topic in the Java world, which can be used for multicasts. This makes such a solution similar to the explicit multicast architecture used in the Hazelcast based prototype.

7 Test Bench and Test Method

To be able to evaluate how the different architectural prototypes fulfill the required QASes for the system in a consistent way, this chapter describes the test bench that has been used for testing the solutions. This includes information on the physical setup, the receipt for running the tests, and finally a description of the method that will be used for measuring and examining the measured results.

7.1 Physical Setup

The deployment diagram in figure 30 on the next page shows the hardware components in the system. Of these the *turbine RFC* and the *OPC server* is of no interest in the scope of this thesis, and since these components only works if they are hooked up to the network bus-system in a physical wind turbine, and it is trivial to simulate sensor readings in the *wind turbine service*, this hardware component has been left out of the test bench. Due to this simulation of sensor readings the requirement that binds the *wind turbine service* to run on a Windows machine also disappears.

Removing these requirements to the test bench, makes it possible to build the entire system using a fully virtualized solution where several virtual machines shares physical hardware. This offers a good price and time tradeoff compared to

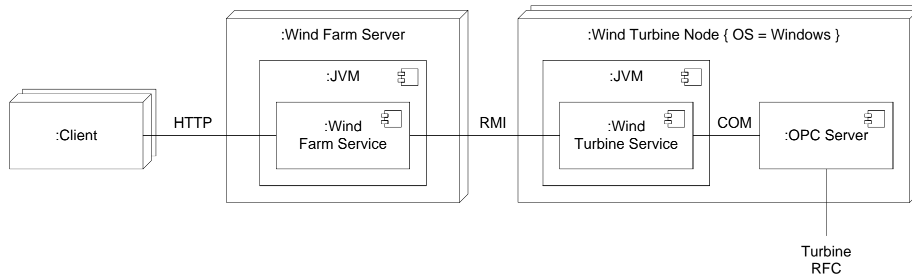


Figure 30: Deployment diagram (same as figure 8 on page 17)

manually building a test laboratory with 18 physical nodes. Today several vendors offers services for running a large number of virtual machines on their server farms, and for this thesis Amazon Elastic Compute Cloud ([Amazon EC2]) has been chosen, using Ubuntu 10.04 instances to host the JVMs running the software.

7.1.1 Number of Nodes

Based on the QASes and the availability theory the number of nodes in the system can be easily calculated to

- 3 wind farm server nodes.
- 10 wind turbine nodes, each with 50 sensors.
- 5 client nodes, each subscribing to 100 sensors (10 from each turbine node).

7.1.2 Practical Problems in Managing the Nodes

The most basic interface Amazon EC2 offers for managing virtual nodes is through a simple web-interface. When managing 18 nodes this interface is cumbersome, but fortunately Amazon EC2 also offers APIs for several languages to manage nodes. This makes it possible to build scripts to launch and upload software to the individual nodes from one central workstation as depicted on figure 31 on the following page.

The central workstation is also used to automate the test setup and test runs in a repeatable way, as it controls both the EC2 nodes and the target applications running on them, using a controller application that makes it possible to start, stop, initialize and fetch log files from the target applications. This is depicted on figure 32 on page 68. The programs and scripts for this are described in details in appendix D and E.

7.1.3 Potential Problems in a Virtualized Solution

Although virtualized solutions such as Amazon EC2 seems optimal for easily testing distributed systems with a large number of nodes, such as the wind farm system, there are certain potential problems that must be considered. This is due to the fact that the overall system when using a virtualized solution seems less deterministic when compared to a tightly controlled number of physical nodes. There are several factors for this:

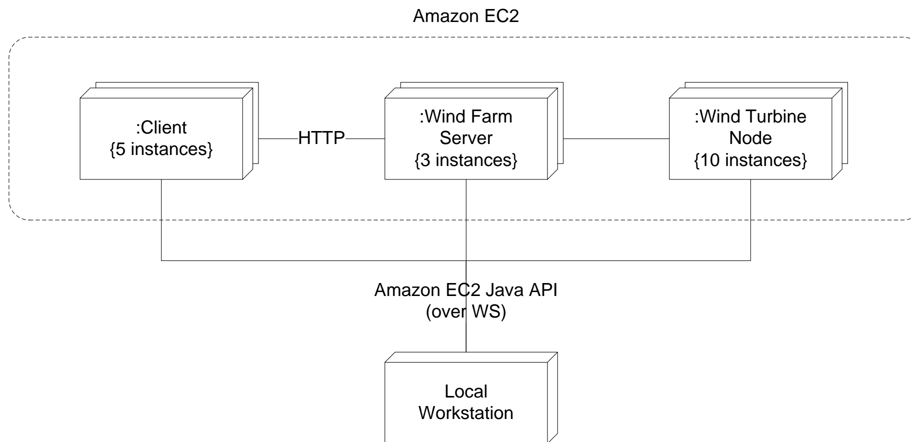


Figure 31: Managing the nodes using a single workstation.

Sharing of Hardware Resources: In a virtualized solution some test runs may share the underlying physical hardware with other virtualized nodes that are highly computational intensive, whereas at other test runs the physical hardware may be almost idle with respect to other virtual nodes.

No Control of Actual Distribution: In a virtualized solution like Amazon EC2 the user does not have any way to control whether the virtual nodes are running on different physical hardware nodes, or whether they are clustered on only a few hardware nodes.

The remedies to avoid these potential problems to skew the test results and conclusions are two fold:

1. The tests must be repeated several times, where the test runs must be distributed over a large time period. Furthermore the tests for the different solutions should be run on the same virtual machines immediately after each other.

2. Measure and record native performance counters that may indicate or reveal factors about the underlying hardware. E.g. measuring network packet round-trip times between the nodes may show that some virtual nodes are clustered in a way that indicates that they are running on the same physical piece of hardware.

There is one academic article, [Ristenpart et al., 2009], that describes mechanisms for detecting whether two virtualized nodes in Amazon EC2 is running at the same physical hardware node, or *co-residence* as it is termed by Ristenpart et al. They describe how Amazon EC2 uses a so called Xen hypervisor as the base “operating system” on the physical nodes, and how the Xen hypervisor reports itself as a network hop in traceroutes. Ristenpart et al. labels this node the Domain0 (Dom0) node.

Based on the description of the Amazon EC2 setup (which it should be said is interpolated from their external measurements, as nothing in the article indicates that the researchers have any inside information from Amazon), they point out 3 different checks for proving co-residence:

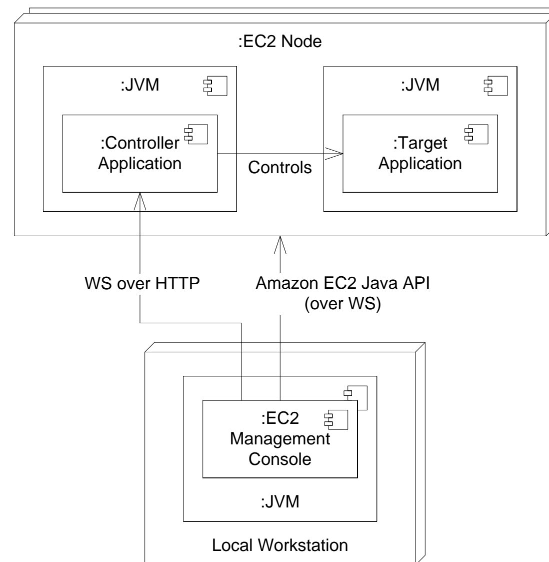


Figure 32: Managing the EC2 nodes and the target applications on them using a standalone controller application running on the EC2 nodes.

1. Matching Dom0 IP address.
2. Small packet round-trip times.
3. Numerically close internal IP addresses (e.g. within 7).

With respect to check number 1, it does no longer seem to be the case that the Dom0 nodes in Amazon EC2 reports as network hops. This also seems plausible as one of the recommendations in the Ristenpart article which takes a security perspective, is that Amazon should disable the Dom0 as reporting as network hops.

With respect to check number 2, Ristenberg et al. does not quantify exactly what "small" is. But nevertheless the relevant information has been recorded and is shown in figure 33 with respectively the number of network hops as reported by traceroute and network round-trip times as reported by the hping tool[19] for one of the test runs.

Analysing the figures does not reveal any critical outliers where it could be expected that two nodes are co-located, although there are 4 pair of nodes where there are no intermediate network hops (i.e. the number in the figure is 1). But there does not seem to be any correlation with extremely low network round-trip times for these 4 pairs of nodes.

Finally check number 3 does not fit the type of EC2 instances, *micro*, used in this thesis, as this type is a new and smaller type than what was present when Ristenpart et al. did there measurements where they used the so called *small* type. So it must be expected that more than 8 virtual nodes of type *micro* shares the same physical node.

[19]Hping is used instead of standard ping, as hping makes it possible to send IP packets instead of icmp packets. And there are indications (but no clear evidence) that Amazon down prioritize icmp traffic in their network.

A final observation from the Ristenpart article is that they conclude that Amazon EC2 does not do any live migration of nodes, meaning that the network layout must be expected to be fixed during the lifetime of a launched instance.

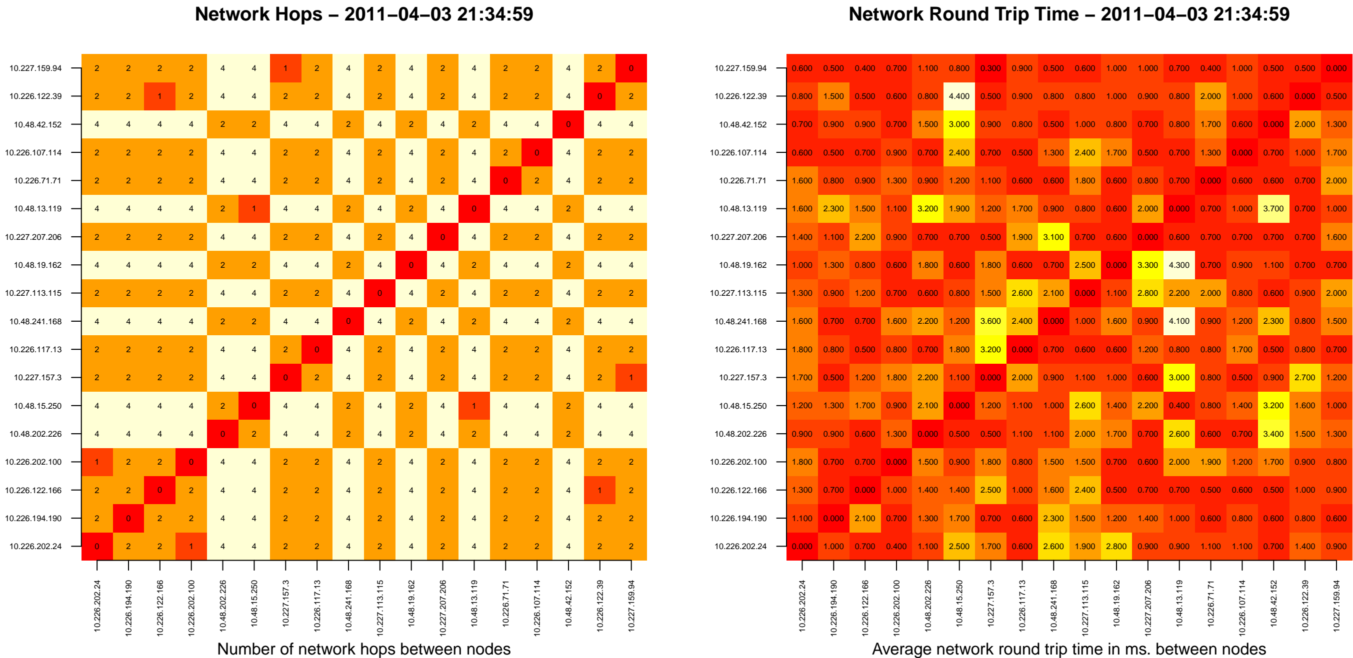


Figure 33: Network measurements between the 18 nodes in the system, depicted as coloured *heat maps* overlayed with the *distance matrix* numbers for one test run.

7.2 Test Receipt

When a system is running in operating mode, crashes will not occur according to a fixed schedule, so testing for whether a system is resistant to this and has the sufficient availability qualities, is an obvious candidate for a mild degree of random or fuzzy testing. In the receipt for testing described below, the last phase, the crash phase, therefore has some randomness build in, as described in the subsection below.

1. Setup phase:
   (a) Launch all wind turbine nodes.
   (b) Launch all wind farm server nodes.
   (c) Launch all clients
2. Normal phase:
   (a) Run for X minutes.
   (b) Collect data from clients.

3. Crash phase:
    (a) Run for X minutes, where wind farm server nodes crashes randomly.
    (b) Collect data from clients.

7.2.1 Algorithm for Random Crashes

The overall idea for the random crashes in the *crash phase* is to let one or more servers crash at random times. This is accomplished by running in "periods" where at the end of a period the following two decisions are made:

1. What wind farm servers must run in the next period.
2. How long is the next period. The upper and lower boundaries for the period length will be decided during the initial test runs.

Both of these decisions should be made based on a standard random number algorithm initialized with a fixed seed so the "random" crashes can be repeated.

To be certain that all test runs can actually be used to evaluate the system QASes a couple of conditions must hold at all times though and thereby limit the randomness a bit:

- At least 1 wind farm server must be running at any point during the test run. If this was not the case this would be a violation of QAS2.
- All wind farm servers must crash at least every fifth period. This protects against the unlikely event where one wind farm server would be running for all periods.

A pseudo code algorithm for modelling the random crashes is given in algorithm 1 on the following page:

Algorithm 1 Pseudo code for random crashes of wind farm servers.

```
/**
 * For each of the 3 farm servers the state in the next period is calculated and
 * returned. The state will be either "up" or "down".
 * The calculated states will satisfy the constraints:
 *   1) The state for each server will maximally be constant for 4
 *       calls to this method.
 *   2) At least one of the server states will be "up".
 *   3) A server will change state with 30% propability.
 */
ServerState[3] determineServerStatesInNextPeriod();

/**
 * Launches all servers that has state "up" in the serverStates arrays, if not
 * already running.
 * The method is synchronous and does not return until the servers are finished
 * launching.
 */
void launchReappearingServers(ServerState[3] states);

/**
 * Stops all servers that has state "down" in the serverStates arrays, if not
 * already stopped.
 * The method is synchronous and does not return until the servers are finished
 * stopping.
 */
void killCrashedServers(ServerState[3] states);

/**
 * Return the length in milliseconds of the next period.
 * This is a random number in the range [20000 ; 40000]
 */
long randomPeriodLength();

/**
 * The actual algorithm for modelling random crashes.
 * Run for 20 rounds of various lengths.
 */
for (int roundsCount = 20; roundsCount < 20; roundsCount++) {
    ServerState[3] serverStates =
            determineServerStatesInNextPeriod();

    // synchronously call, wait til up
    launchReappearingServers(serverStates);

    // synchronously call, wait til down
    killCrashedServers(serverStates);

    sleep(randomPeriodLength());
}
```

7.3 Measurement Method

7.3.1 Data to Collect from Clients

To be able to verify that the QASes hold each client must with 1 second intervals record the “age” of the data for each sensor that it subscribes to. Where “age” is defined as the number of seconds since the data was sent from the wind turbine node.

The easiest way to record this is to let the wind turbine nodes attach a timestamp to each sent value, and use a local NTP server to synchronize the clock of the nodes.

7.3.2 Analyzing Data

The collected data series should be analyzed based on a plot of the maximum data age as depicted in the simplified and stylized figure 34. For normal periods with no crashes the data age should fluctuate around 1 second to fulfill QAS3. At crash times the data will grow older as seen from $t = 7$ to $t = 11$ on the figure, indicating a server crash at $t = 7$ and a reconnect to another server between $t = 11$ and $t = 12$. At these crash times the data age must not exceed 5 seconds to fulfill QAS1.

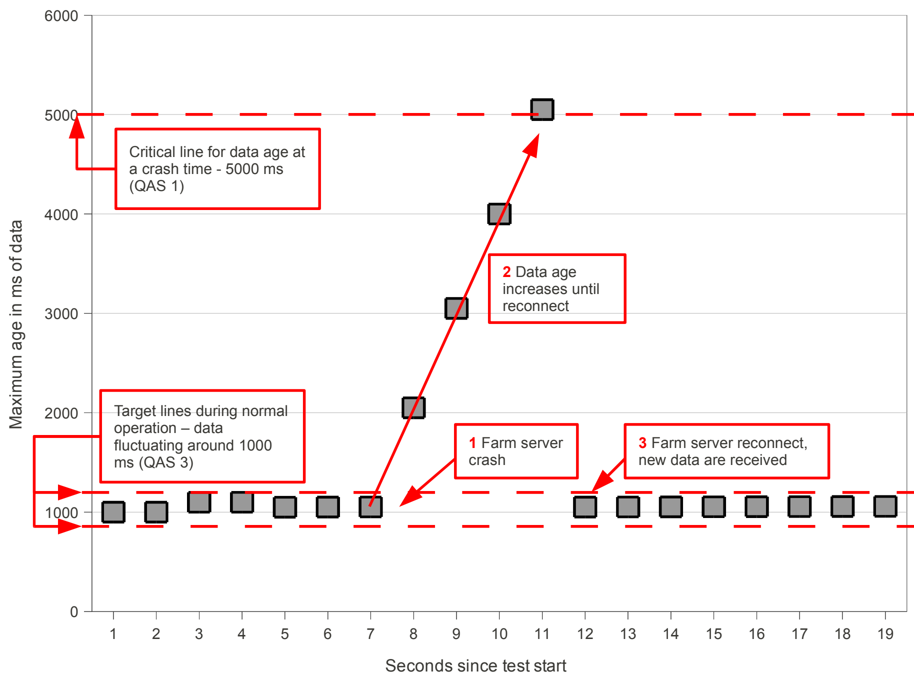


Figure 34: Stylized example plot of maximum data age on a client.

The actual data series measured will show longer periods of data and will therefore be depicted as connected lines instead of individual points. Furthermore it will show the data for all 5 client nodes in the same diagram, as well as the crash and restart times of the 3 farm server nodes. A commented actual measurement is shown in figure 35 on the following page.

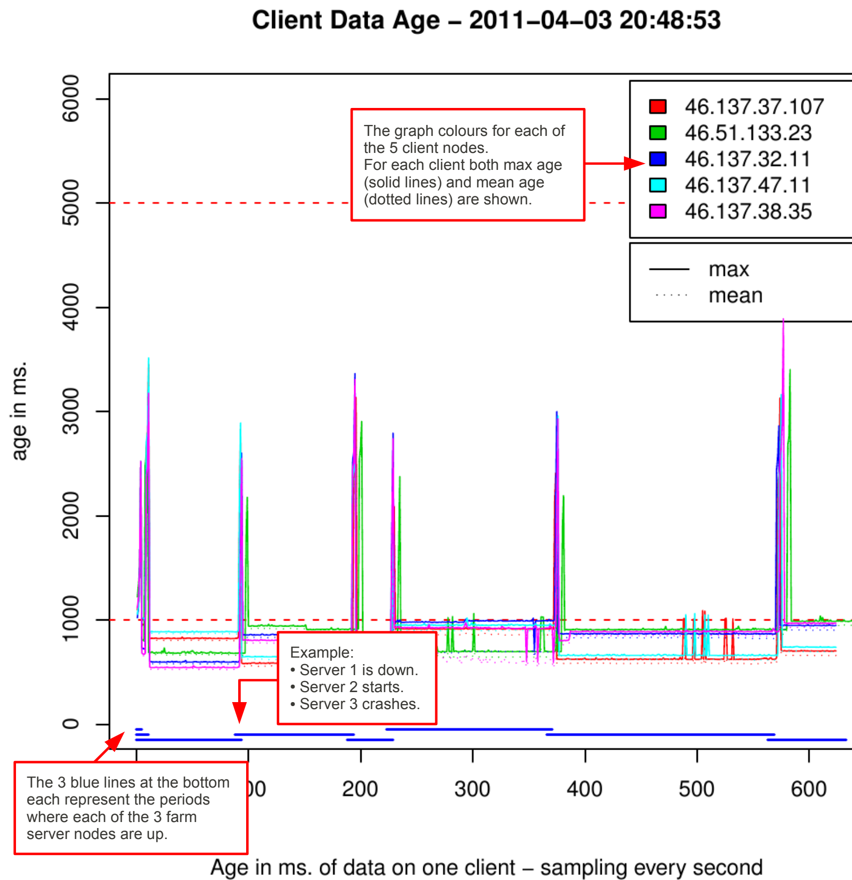


Figure 35: Commented plot with actual measured maximum and average data age for 5 clients and running periods for the 3 farm servers.

7.4 Alternative Measurement Method - Qualitative Network Packet Analysis

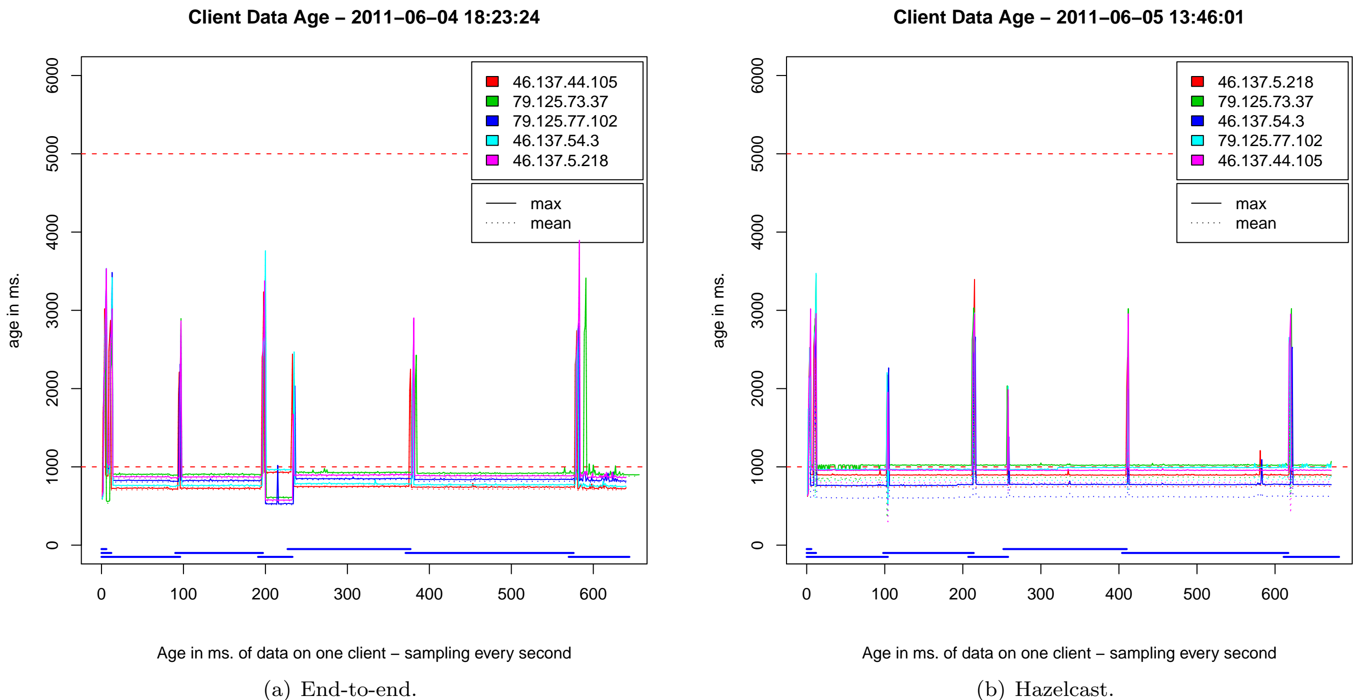
At the offset of the thesis it was considered to do a detailed qualitative network packet analysis of the different solutions as a supplement to the quantitative hard measurements on the observed timings as described in the previous chapter. As the quantitative measurements seems to be a good way to verify whether the QASes was fulfilled, the detailed packet analysis was dismissed. Initially the analysis was performed for the original system using pure RMI communication though. For the interested reader, this analysis is found in appendix F.

8 Evaluating the Prototypes

To evaluate the 3 prototypes, each one was first developed and sporadically tested locally on the development PC. After this removed the visible problems, the test bench was used for further testing and remote debugging. As the debug-

ging on a system with 18 nodes, where 3 of them (the farm servers) randomly crashes, is not really suited for visual debugging in a tool like Eclipse, this was mainly done by observing log files as they were written on the nodes.

To get reasonable evidence that each prototype actually fulfills the QASes, each of the successful prototypes (the Hazelcast and the end-to-end based prototypes) was scheduled for 100 test runs with 20 periods in each run. This means that each of the prototypes was tested over a time period that came close to 24 hours. These two sequences of 100 test runs was both primed with the same seed in the random number generator used in the randomization of period lengths and crashes described in algorithm 1 on page 71. This means that the crash sequences are repeatable and are the same for the two tests, so there should not be any of them having a “lucky” or “easy” sequence of crashes. Furthermore the network layout was the same during the test runs as the same set of Amazon EC2 nodes was used.



(a) End-to-end.

(b) Hazelcast.

Figure 36: Test run 1 for the end-to-end based prototype and the Hazelcast based prototype.

Figure 36 shows test run number 1 for the end-to-end and the Hazelcast based prototypes, where it should be easily visible from the 3 blue server uptime lines at the bottom of the graphs that the crash sequence and period lengths are identical. The observant reader will notice slightly longer period lengths for the Hazelcast test run though, which is due to the fact that the Hazelcast prototype is a little slower in starting and stopping the crashing farm server nodes, and as this is added to the period lengths this leads to this small mis-alignment. This is not believed to be of any significance in the tests though.

In all truthfulness it should be pointed out, that neither the selection of exactly 100 test runs, the 24 hour test period, or any other of the comments in the above sections are based on a theoretical statistical foundation. So the reader should not go away believing that the results are based on proof by statistical evidence, as a professional statistician could probably find problems with the method used.

Besides the recording of the formalized test sequences a visual inspection of the network load on the 3 types of nodes was performed during one test run for each prototype. This was done using the Linux `ifstat` command line tool, using the exact command `ifstat -n -t -b` which reports the network load in Kbits/sec every second.

In the sub chapters below follows detailed comments to the results for each of the 3 prototypes.

8.1 The Terracotta Based Prototype

Running the passive redundancy DSM-based Terracotta prototype described in chapter 6.2 on the test bench, was quite disappointing. Actually it was so bad that the prototype was not even able to fulfill the basic 1 second deadline in QAS 3 when all servers was running and no crashes occurred at all. Figure 37 shows the measurements for two of these test runs, where it can be seen that all 3 farm server nodes are running throughout the entire test.

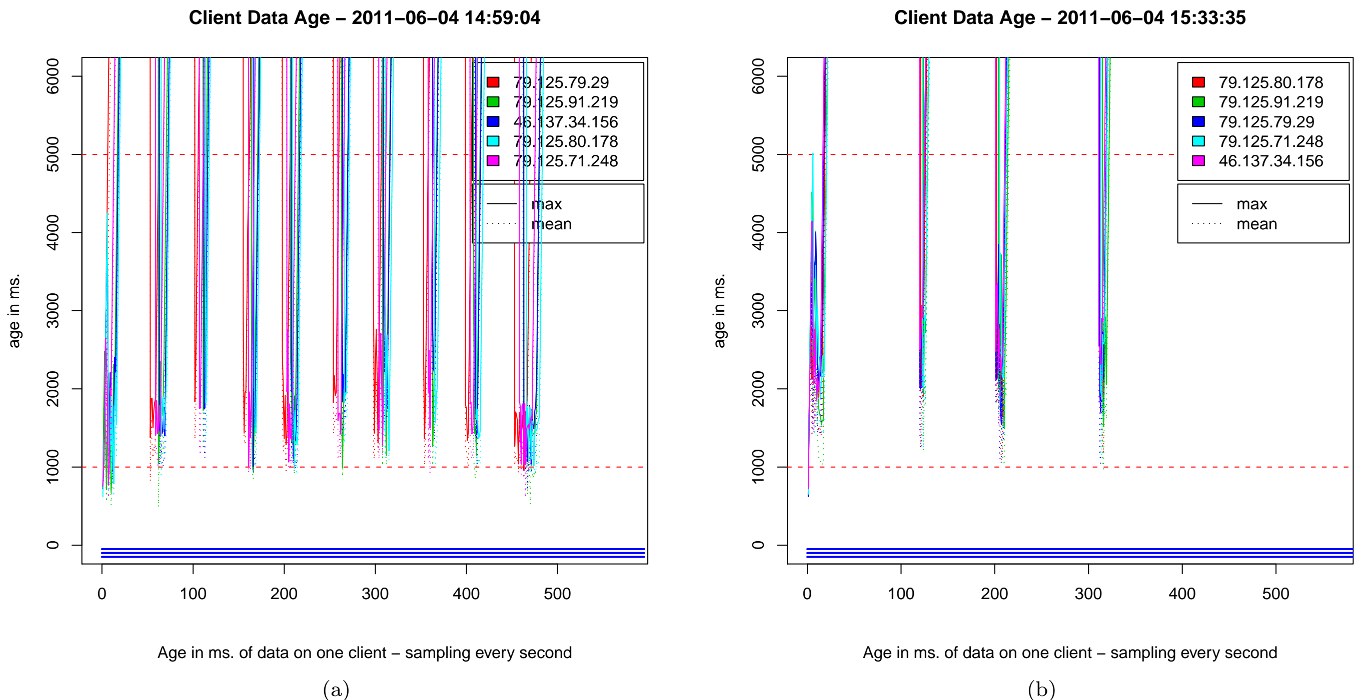


Figure 37: Two measurements for the Terracotta based prototype.

Two potential reasons that might explain the patterns seen on figure 37

could be:

- Delayed flushing of data in the DSM. This would explain the reoccurring spikes if data written locally to the DSM is only flushed to the other nodes once every 30th second, which seems to be the interval on graph 37(a). But this observation is not consistent with the pattern seen in the second run depicted on graph 37(b).
- Mismatch between the passive solutions used in the internal versus the external group. The Terracotta based prototype uses a *global* active farm server node in the internal group, which might lead to problems since the clients in the external group randomly selects different farm server nodes as their *local* active farm server node. Therefore some of the client nodes might be using one of the passive farm server nodes, which will lead to significant delays if the Terracotta implementation is done in a way where all reads hits the active farm server node.

It should be said that when the prototype ran with one instance of all 3 nodes on the development PC the observed values was well below the 1 second deadline. This fact made it somewhat harder to debug on, as the programming logic in the prototype seemed to work fine in the small scale.

Although several debugging sessions were performed, and different attempts to tune the Terracotta settings was done, none of these brought any significant changes in a positive direction. As a last resort the prototype was even changed to use the explicit Terracotta Toolkit API ([Terracotta, 2011, ch.7]), and still with no progress.

With respect to the measured network load during the test runs as reported by `ifstat`, this is shown in table 5. When comparing these numbers with the observed loads for the two other prototypes, it is noted that for the farm server and turbine node the numbers are more than a factor 10 larger for the Terracotta prototype, and shows extreme fluctuations as seen from the large peak numbers. For the client node the Terracotta numbers are close to half the numbers observed for the two other prototypes, which can simply be explained by the fact that almost no data gets through to the client nodes.

Node Type	**Kbps/sec in**	**Kbps/sec out**
Farm server	6000 (peak at 40000)	10000 (peak at 75000)
Turbine node	150 (peak at 300)	600 (peak at 9000)
Client	120	12

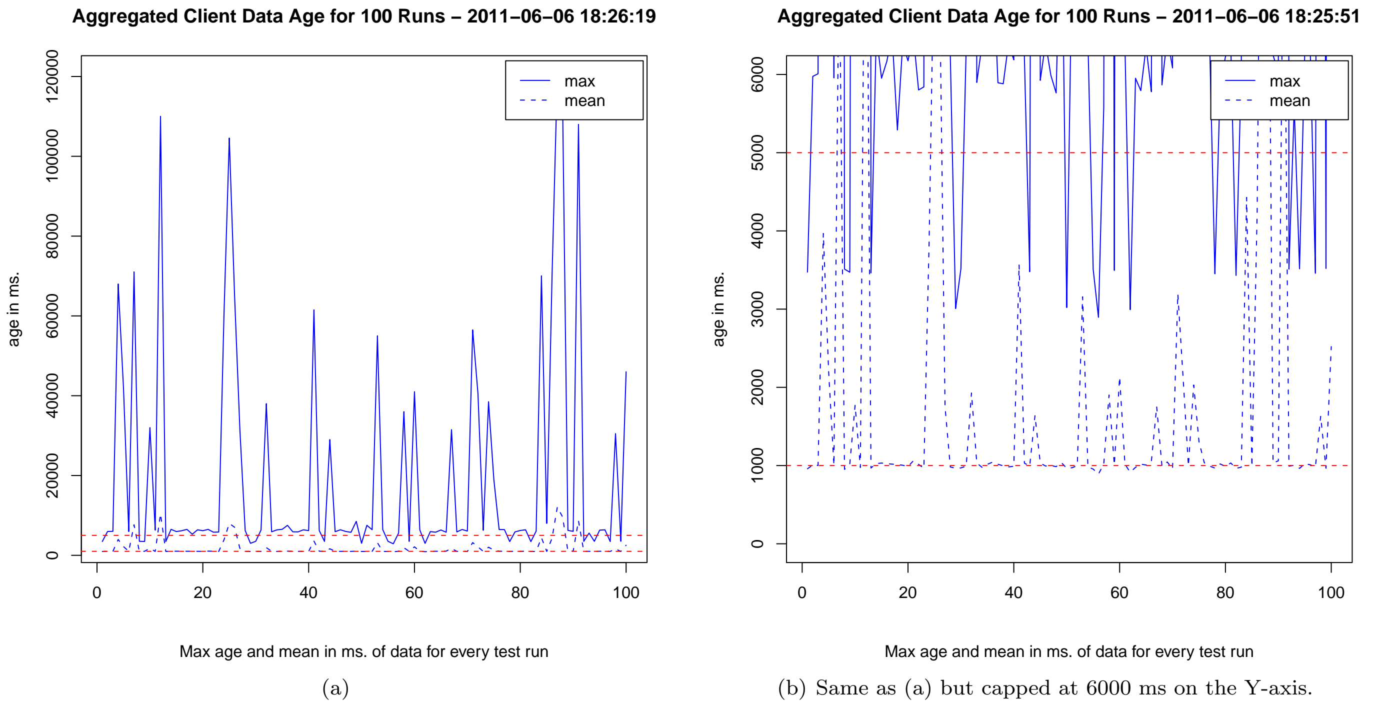
Table 5: Network load per node type for the Terracotta based prototype

So the conclusion with respect to the Terracotta based prototype, is that it was a complete failure, as it did not fulfill any of the three QASes. Therefore no extensive testing with 100 test runs was performed for this prototype. Furthermore it was seen that a very large overhead of network traffic was generated by the Terracotta servers.

8.2 The Hazelcast Based Prototype

The Hazelcast based prototype at first seemed to be successful on delivering on all 3 QASes during the development and the initial individual test runs on the

test bench. But as it can be seen on figure 38 the test over 100 test runs showed that the reality was not as bright. The figures are a summary of all 100 tests in one graph, showing the *maximum* observed client data age for each of the 100 test runs as the solid line, and the *mean value over the max values* as the dashed line.



(a)

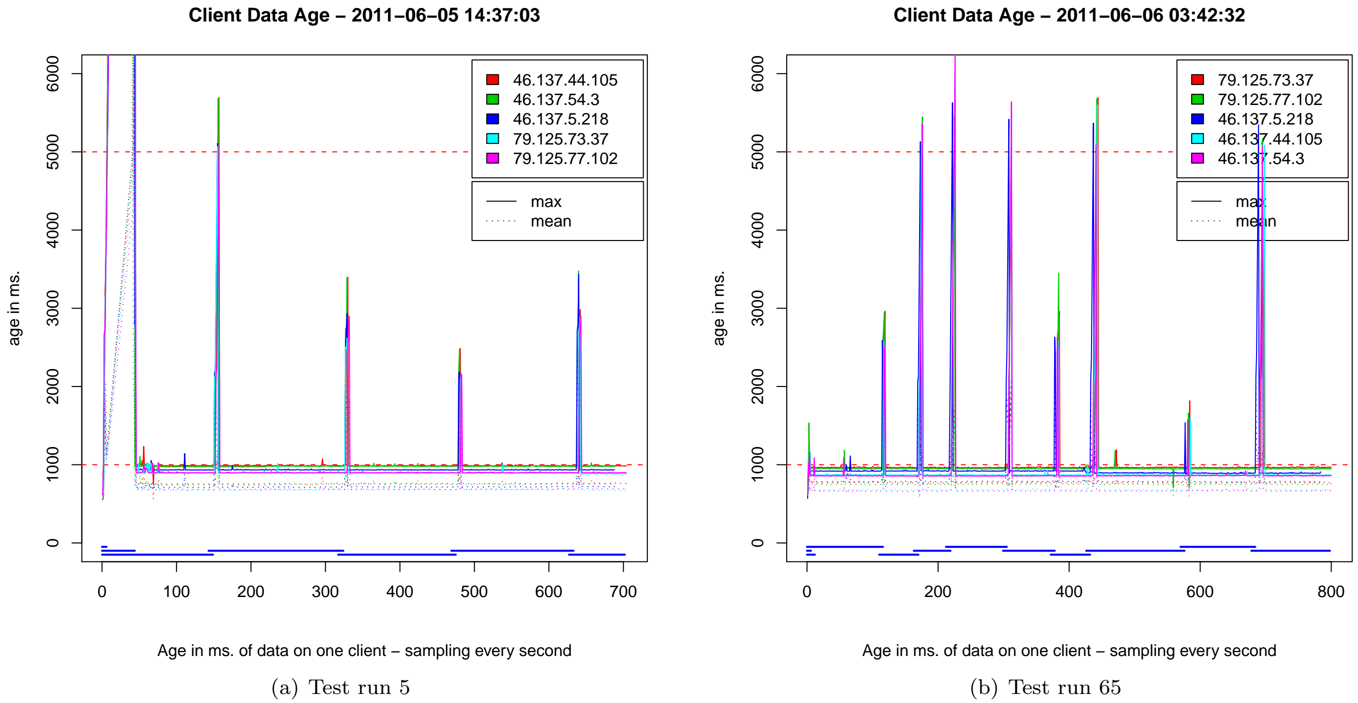
(b) Same as (a) but capped at 6000 ms on the Y-axis.

Figure 38: Aggregated max and mean of the max values for all 100 test runs for the Hazelcast based prototype.

The two figures shows that only 20% of the test runs was successful in that the maximum observed value for the test run was below the 5 seconds deadline of QAS1. For the remaining 80% of the tests there were at least one failover situation per test that failed in fulfilling QAS1.

Zooming in on the individual test runs to find an explanation for these aggregated numbers shows that the successful tests were all similar to the graph previously depicted in figure 36(b) on page 74. For the failing test runs there were two typical kinds of errors visible, depicted in figure 39(a) and (b). The (b) graph showing test run number 65 shows a situation where the 5 second failover deadline is slightly exceeded at some of the farm server crashes. This is definitely a fail with respect to QAS1, but not at all comparable to the way the Terracotta based prototype failed.

The second observed type of failure is shown on (a) showing test run number 5 where the prototype almost breaks down at the beginning of the test. In the most extreme of these failures the data age reaches almost 110 seconds before falling back to 1 second. This type of failure was only observed at the beginning of the test runs.



(a) Test run 5

(b) Test run 65

Figure 39: Two measurements for the Hazelcast based prototype.

A more detailed analysis of this second type of failure is given in the full test report for the Hazelcast based prototype in appendix G, which is found in the separate document master2011-MA-appendix-G.pdf. The conclusion from that analysis is that there seems to be test runs where 2-3 of the 10 turbine nodes fails in joining the Hazelcast group at the startup of the tests. This is based on the fact that the detailed analysis show that most sensor values do get through to the clients.

The network load during the test runs is shown in table 6. From these numbers it is clear that the network traffic is significantly smaller than the load reported for the Terracotta prototype. In general the numbers are similar to the traffic measured for the end-to-end prototype, except for the farm server out measurement which is is approximately 25% larger for the Hazelcast prototype, and the turbine node in measurement which is a factor 7 larger for the Hazelcast prototype.

Node Type	Kbps/sec in	Kbps/sec out
Farm server	570	1600
Turbine node	41	47
Client	245	24

Table 6: Network load per node type for the Hazelcast based prototype

During the test run the management console running on the local workstation

crashed after test run number 17. Therefore the test was restarted at that point for the remaining 83 tests. When the test runs was later analysed it was noticed that this restart poses a statistical problem as the seed for the restarted sequence was also reset, wherefore test number 1 and test number 18 is actually a test of the same crash sequence (and so on up until test number 17 and 34). Seen from a straight statistical view the Hazelcast test should be rerun to be completely comparable to the end-to-end test run. But from a pragmatic view it is not assumed that this would lead to any change in the conclusion about the performance of the Hazelcast prototype.

So summarizing, there are at least two problems with the Hazelcast prototype that stops it from fulfilling the QASes:

1. Occasionally smaller exceedings of the 5 second deadline. This problem might be hard to solve, as it is not expected to be a direct bug in the source code, but more a performance problem with the Hazelcast multicast algorithm.

2. Problems at startup for some tests, leading to significantly exceeding the 5 second deadline for a prolonged period. This problem is probably attributable to a bug in the source code. With some effort it is assumed that it would be possible to find and fix this.

8.3 The End-to-End Based Prototype

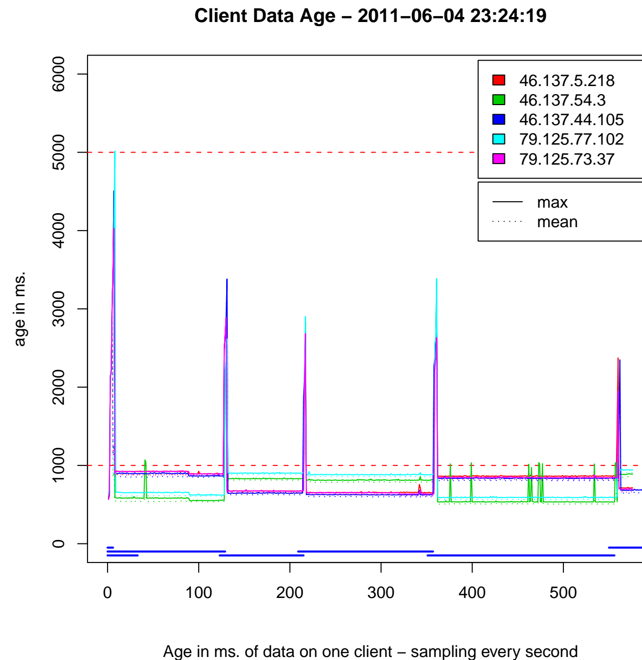


Figure 40: Test run 27 for the end-to-end prototype.

The test runs for the end-to-end based prototype, showed that the prototype was successful in fulfilling all 3 QASes. With respect to the 5 second deadline

at failover time, all tests except test run number 27 shown in figure 40 was well below this. For this single test the deadline was exceeded with 15 ms, meaning that on one client a maximum data age of 5.015 seconds was seen. Since this is a very small surpassing of the limit, and since this is the only occurrence out of approximately 2500 failovers during the 100 runs, it is simple not sufficient reason for dismissing the prototype.

As for the 1 second soft deadline during normal operation this is also in overall respected by the prototype. There are small glitches here and there, but none that exceeds the limit in a problematic way.

The overall success of the end-to-end based prototype is seen in figure 41. The figure depicts the *maximum* observed client data age for each of the 100 test runs as the solid line, and the *mean value over the max values* as the dashed line. It is clearly seen that test run number 27 is a single outlier, and that for all other tests the maximum age is in the range 3.5 to 4.0 seconds. Furthermore the mean values are slightly below the 1 second deadline.

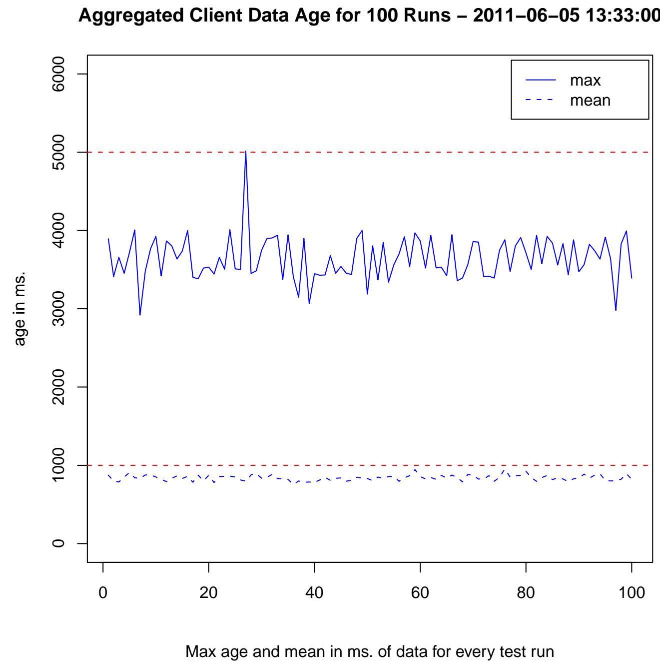


Figure 41: Aggregated max and mean of the max values for all 100 test runs for the end-to-end prototype.

The network load during the test runs is shown in table 7. Comparing these numbers to the two other prototypes it is clear that the network traffic is significantly smaller than the load reported for the Terracotta prototype, and slightly smaller than the Hazelcast load for the farm server out and turbine node in measurements.

The full test report showing the detailed graphs for all 100 tests runs for the end-to-end prototype can be found in appendix H, which is found in the separate document master2011-MA-appendix-H.pdf.

Node Type	Kbps/sec in	Kbps/sec out
Farm server	590	1250
Turbine node	6	48
Client	245	24

Table 7: Network load per node type for the end-to-end prototype

9 Conclusion

This thesis has described how the theory on availability can be applied to a specific distributed wind farm SCADA system. Based on this 3 architectural prototypes were build, and the performance of these with respect to the architectural QASes from chapter 2.5.2 was evaluated using a well defined test bench build using virtual machines running on Amazon EC2.

The first prototype was a *passive redundancy* solution using *DSM*, and was build using the software product [Terracotta]. The evaluation of this prototype showed that it failed severely with respect to even basic performance when the system was running in normal mode. Furthermore a significant overhead in network traffic, when compared to the two other prototypes, was observed. Finally the techniques used in this prototype required significant changes in the original system as the communication patterns was changed from explicit network communication to hidden communication using the DSM. So concluding, this solution failed severely, and although it might be possible to get the prototype to deliver on the QASes by turning the correct knobs in the Terracotta software, it does not feel like a natural fit to solve the problem using DSM.

The second prototype was an *active redundancy* solution using *network multicast communication* build using the software product [Hazelcast]. After the first informal tests it seemed that this prototype solved the QASes fine. Furthermore the network traffic measured for this prototype showed significantly less traffic than the Terracotta based prototypes, and it was also easier to build into the existing system. A formal long-running test on the test bench uncovered some severe problems with the prototype at the failover situations. This result showed the strengths of building a well defined test bench to evaluate solutions, even though it is of course harder to build a test bench for a non-deterministic distributed system with many nodes. Although the Hazelcast based prototype failed in fulfilling the QASes, it is believed that the problems should be solvable, and the well defined test bench makes it easy to verify whether a proposed fix actually solves the problems.

The third prototype used the so called *end-to-end principle* with only *re-computable state* (also known as "soft state") on the critical central wind farm server nodes. This made it possible to build a prototype without using neither the passive nor the active redundancy tactic, and still fulfill the QASes. Running this prototype on the test bench verified that this third prototype outperformed the two other prototypes and clearly fulfilled the QASes. With respect to the observed network traffic this prototype was observed to be slightly more efficient than the Hazelcast based prototype. The prototype has one significant drawback though, as it is more complex in the communication concepts and implementation than the two other prototypes, making it harder to evolve over time in a safe way.

An important byproduct of the main work in this thesis, is the well defined test bench using a virtualized environment. This should with only minor modifications be usable for evaluating similar distributed problems in the future.

Appendices

A Table of Contents of the Source Archive

Together with this thesis a zip archive containing source code for the prototypes, appendix G and H and other artifacts is delivered. The archive is named master2011-MA-source.zip. The contents of the archive is enumerated below.

A.1 Source Code for the Prototypes

The source code for the prototypes and the test bench tools are included in the source folder. It should be noticed that some large 3. party dependencies are not included in the archive. These should be either manually downloaded as described in the source/readme.txt file, or automatically downloaded by using the shell file source/setup.sh.

The sub folders in the source folder are listed below:

model: This folder contains source code and build files for the original model system with no availability functionality as described in chapter 2.

terracotta: The source code and build files for the Terracotta based prototype described in chapter 6.2 is present in this folder.

hazelcast: The source code and build files for the Hazelcast based prototype described in chapter 6.3 is present in this folder.

end2end: The source code and build files for the end-to-end based prototype described in chapter 6.4 is present in this folder.

aws: This folder contains shell script files automating the task of preparing and creating an Amazon EC2 image that can be used as node in the test bench. The content of this folder is described in details in appendix E.

ec2-mgmt: This folder contains the source code for the EC2 Java based controller applications used for running the tests. This is described in details in appendix D.

A.2 Results of the Test Runs

During the test runs for the prototypes, the age of the sensor readings observed at the client nodes was emitted to plain text csv-files. These files are included in the archive in the test-runs folder.

The analysis of these data for the Hazelcast and the end-to-end based prototypes are included as the two appendices G and H, placed as pdf files in the root of the archive. The highlights from these analyses has already been described in chapter 8.

A.3 Other Artifacts

Appendix F describes a disassembly of the RMI communication taking place between the nodes in the model system. The raw network capture files for this analysis is found in the rmi-analysis folder in the archive.

B Building and Running the Software

This appendix contains a recipe for setting up, building and running the tests. It is not pinned out in extreme details and requires some familiarity with build files, Java and Amazon EC2.

B.1 Prerequisites

The prototypes and tests was build and controlled from a Ubuntu 10.04 desktop with the software below installed.

- Java 1.6.
- [Gradle] build tool[20].
- [GNU R] statistical library for creating the graphs.
- Bash shell for running the scripts described in appendix E.

The Java parts of the prototypes should of course be usable on all platforms supported by Java, but the helper scripts, as well as the R scripts launched from the ec2-mgmt-console program might require small changes if they should run on e.g. Windows (cygwin will of course help).

B.2 Download 3. Party Libraries

Three of the source projects (aws, hazelcast and terracotta) does not contain all of the required artifacts, as the 3. party libraries used by these are rather big.

These dependencies should therefore be either manually downloaded and extracted as described in the source/readme.txt file, or automatically downloaded and extracted by using the shell file source/setup.sh.

B.3 Setup an Amazon EC2 Account

To be able to run the compiled prototypes on the Amazon EC2 test bench it is necessary to create an Amazon EC2 account. At the time of writing Amazon offers free usage for a year of so called micro instances at `https://aws.amazon.com/free/`.

It is of course possible to build and run the prototypes in small locally without an Amazon EC2 account.

As a part of creating the account it is necessary to setup several types of access credentials used by different parts of the Amazon infrastructure. This is done in the security credentials part of the account management at `https://aws.amazon.com/account/`. It might be preferable to read appendix D and E first to get an understanding of what these security credentials are used for.

The Amazon access key id and secret part should be entered into the two files:

- ec2-mgmt/ec2-controller/src/main/java/dk/accel/misw/mp/ec2/ctrl/impl/AwsCredentials.properties

[20] Gradle was selected instead of the more mainstream Ant or Maven build tools, mainly because these prototypes was a good isolated project to get some experience with a new build tool.

- ec2-mgmt/ec2-mgmt-console/src/main/java/dk/accel/misw/mp/ec2/AwsCredentials.properties

A X.509 certificate will also need to be created. The public and private parts of the certificate should be stored in the aws/certs folder and be referenced from the aws/ec2-env file.

Finally a Amazon EC2 key pair for ssh access must be created. The name of this must be inserted into both the aws/ec2-env file and the two AwsCredentials.properties listed above. The private key for this should preferable be downloaded into the aws/ssh folder. It could of course also be put into the normal private ~/.ssh folder, but the scripts in the aws folder have been configured to avoid interfering with the normal .ssh keys and known_hosts file.

Finally the Amazon EC2 user id (without dashes) should be entered into the two AwsCredentials.properties listed above.

B.4 Create an Amazon EC2 Ubuntu 10.04 Image

This step is described in appendix E, but before doing this it is necessary to build the ec2-controller application that will be uploaded to the Amazon image. This is done by running the Gradle command below in the source/ec2-mgmt folder:

```
gradle assemble
```

B.5 Building Using Gradle

For building the individual prototypes the following Gradle commands can be used in each of the source/<prototype> folders:

gradle assemble: Builds and assembles all three components for a prototype.

gradle clean: Cleans all components for a prototype.

gradle eclipse: Creates eclipse project files for a prototype. Note that each of the prototypes contains components with the same names (e.g. client, server, etc.), so a separate Eclipse workspace must be used for each prototype.

gradle cleanEclipse: Removes the eclipse project files.

To speedup Gradle it is beneficial to use the −−daemon option to launch the Gradle daemon process. The full syntax is then like:

```
gradle --daemon assemble
```

B.6 Eclipse Launch Files

The individual prototypes as well as the test bench program contains a list of Eclipse launch files in the etc folder of each component. These should automatically appear in the run -> run configurations menu in Eclipse when a project has been imported.

These should be relatively self describing given access to the source code.

C Manual for Reading the Source Code

All three prototypes as well as the original model system has the same source code layout. Each contains the following four modules:

common: Contains some shared utility classes.

client: Contains the source code for the clients.

node: Contains the source code for the wind turbine service.

server: Contains the source code for the wind farm service.

In the sections below the components from the C & C diagram for the wind turbine service and the wind farm service are mapped to source code classes.

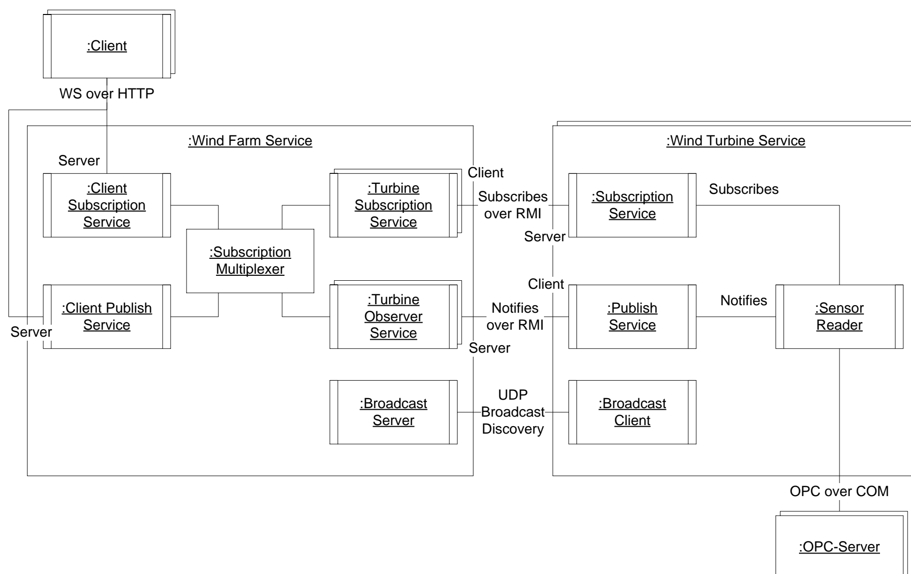


Figure 42: Component and connector diagram (previously shown in figure 5 on page 14).

C.1 Wind Turbine Service Source Code

The four main components from the C & C view for the turbine service maps to the source code as follows:

Subscription Service: Is implemented by the dk.accel.misw.mp.model.node.impl.NodeSubscriptionServiceImpl class.

Publish Service: Is also implemented in the dk.accel.misw.mp.model.node.impl.NodeSubscriptionServiceImpl class, by the inner class TurbineSubscription.

Sensor Reader: Is implemented by the dk.accel.misw.mp.model.node.mock.SensorMock class.

Broadcast Client: Is implemented by the dk.accel.misw.mp.model.node.impl.BroadcastClientImpl class.

C.2 Wind Farm Service Source Code

The six main components from the C & C view for the wind farm service maps to the source code as follows:

Client Subscription Service: Is specified by the dk.accel.misw.mp.model.server .client.ClientSubscriptionSvc interface and implemented by the dk.accel. misw.mp.model.server.impl.ClientSubscriptionSvcImpl class.

Client Publish Service: Is specified by the dk.accel.misw.mp.model.server.client .ClientPublishSvc interface and implemented by the dk.accel.misw.mp.model .server.impl.ClientPublishSvcImpl class.

Subscription Multiplexer: Is implemented by the dk.accel.misw.mp.model. server.impl.SubscriptionMultiplexerImpl class and some of the other classes in the dk.accel.misw.mp.model.server.impl package.

Turbine Subscription Service: Is implemented by the dk.accel.misw.mp.model .server.impl.TurbineSubscriptionSvc class.

Turbine Observer Service: Is implemented by the dk.accel.misw.mp.model. server.impl.TurbineObserverSvcImpl class.

Broadcast Server: Is implemented by the dk.accel.misw.mp.model.server.impl .BroadcastServer class.

D Java Program for Automating the Test Runs

The source archive for this thesis contains the java programs for automating the test runs on Amazon EC2 in the source/ec2-mgmt folder. This folder contains two modules:

ec2-mgmt-console: This module contains the source code for the management console running at the developers local host. This is basically a command line tool accepting one of five commands { start, stop, probenet, runtests, fulltests } for controlling the EC2 nodes and the software running on them. The five commands are individually described below in section D.1.

ec2-controller: This module contains the source code for the controller application running on the remote EC2 nodes. This basically just exposes a simple webservice API with the five operations described in section D.2. This component is packaged on the EC2 image as described in appendix E, and is set to auto start when an EC2 instance is launched. The webservice API it exposes makes it possible to upload new target application images, and start and stop the target applications remotely using the management console.

The overall runtime interoperation of these components is shown in figure 43 on the next page.

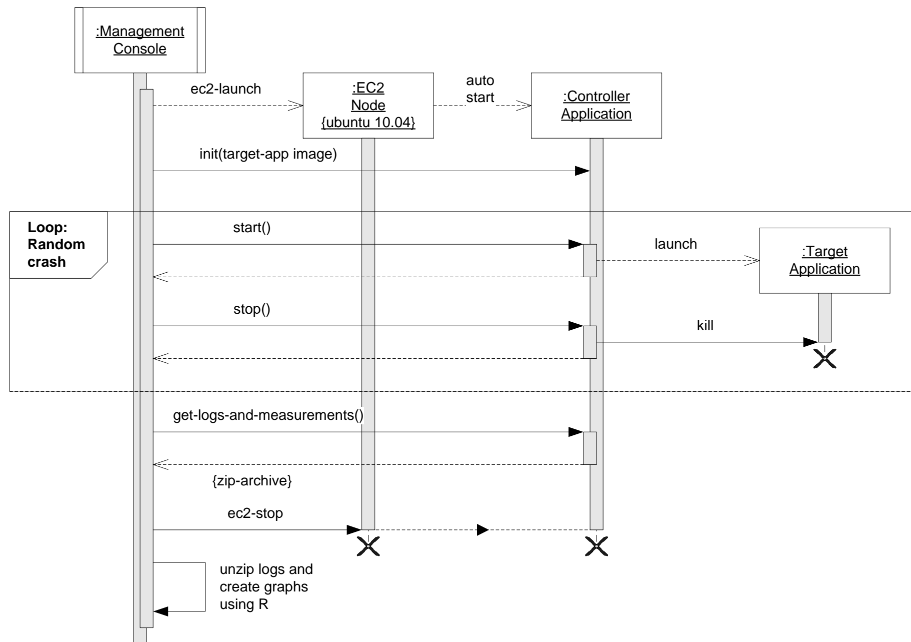


Figure 43: Sequence diagram showing how the *management console* instantiates EC2 nodes and controls the target applications using the *controller application* running on the EC2 node.

D.1 Management Console Commands

The management console application dk.accel.misw.mp.ec2.Main accepts the following 5 commands, for which Eclipse launch files are included in the project in the ec2-mgmt-console/etc folder:

start: This command launches 18 EC2 nodes using the Amazon EC2 API, and writes the IP-addresses of these to a hidden .instances.bin file.

stop: This command reads the node addresses from the .instances.bin file and stops the nodes it finds therein.

probenet: This command reads the node addresses from the .instances.bin file and sends a webservice call to each of them to probe the network with respect to round-trip times and number of network hops to the 17 other nodes. Each node returns a list of the results for this, and the management console uses a small [GNU R] script to generate graphs for the network measurements in the format seen on figure 33 on page 69. The R scripts are present in the ec2-mgmt-console/etc/r-scripts folder.

runtests: This command starts by uploading images for the 3 different target application types, farm servers, turbine nodes and clients for one prototype to the EC2 nodes by using the controller application init() method. Thereafter the test algorithm described in algorithm 1 on page 71 is initiated. Finally the logs and measurement data are retrieved from the EC2 nodes and another small R script generates graphs for the sensor data age

for the 5 clients in the format seen on figure 35 on page 73. The command requires the -Dtest-project environment option to be set and point to the folders containing a compiled version of the prototype to test.

fulltests: This command is similar to the runtests command, but performs 100 repetitions of the test runs.

D.2 Controller Application Webservice Operations

The controller application exposes the following 5 webservice operations:

```
/**
 * Download the java program from Amazon S3.
 */
void init(String s3key, Environment env);

/**
 * Probe the network to the hosts in the ipAddresses parameter.
 */
NodeNetworkInfoList probeNetwork(List<String> ipAddresses);

/**
 * Start the downloaded java program.
 */
void start();

/**
 * Stop the download java program.
 */
void stop();

/**
 * Zip the log folder and upload it to S3.
 *
 * @return the s3key for the data.
 */
String uploadData(String s3prefix);
```

E Scripts for Automating the Test Bench Setup

The source archive for this thesis contains a set of bash scripts automating the setup of an Ubuntu 10.04 based Amazon EC2 image that can be used for launching the 18 nodes in the test bench setup. These scripts are found in the source/aws folder. The main files in this folder are listed below.

Before running these scripts the ec2-controller application must have been configured with EC2 credentials in the ec2-controller/src/main/java/dk/accel/misw/mp/ec2/ctrl/impl/AwsCredentials.properties file, and must have been com-

piled and packaged by running the Gradle command below in the source/ec2-mgmt folder:

```
gradle assemble
```

The files and folders in the source/aws folder are:

ec2-env.template: This file is a template for the file storing the EC2 user account credentials. A copy of this file should be created and named ec2-env, where after the EC2 certificate and SSH keypair name should be specified in the file.

ec2-image-create.sh: The main script file that creates the EC2 image. The steps it performs are:

1. Launches a plain EC2 Ubuntu 10.04 image.
2. Installs a Java VM in it.
3. Uploads the ec2-controller application to it, and configures it for auto-starting when the node starts.
4. Perform other minor system maintenance tasks on the system, such as set up firewall, update packages and install some used system tools (such as hping3, zip, unzip, ifstat, sysstat).
5. Create a new private EC2 image based on the modified image, which can now be launched in 18 instances for the tests.

remote/setup.sh: A script that is uploaded by the ec2-image-create.sh script to the running EC2 image, and executed remotely using plain SSH with the command:

```
ssh -t -F ./ssh/config ${HOST} sudo bash ./setup.sh
```

This script performs the parts of the ec2-image-create.sh that is to be executed on the remote node (i.e. step 2, 3 and 4 in the above list).

scripts-folder: This folder contains some helper functions used by the ec2-image-create.sh script.

F Qualitative Analysis - RMI Packet Analysis

Early in the thesis work it was considered to do a detailed network packet analysis as a supplement to the quantitative timing measurements.

The idea behind such an analysis was that the number of network packets involved in the communication between the farm server and a turbine node would be a good indication of how much overhead the different prototypes would introduce when solving the availability problem in different ways. In the following sections the analysis that was done at the outset of the thesis work, on the original baseline system, is described.

F.1 Hypothesis on Packet Numbers and Sizes

For the baseline system with its pure RMI solution it would be expected that the number of network *packets* will follow the trivial linear mathematical function: $y = Mx + R_c$, where x is the number of RMI calls, M is the network packet count per RMI call, and R_c is fixed overhead related to RMI housekeeping, e.g. for distributed garbage collection. Furthermore the *size of the individual packets* should also be measured. This is expected to be a function related to the number of sensors that are subscribed: $Z = R_1 s + R_2$, where s is the number of sensors that at least one client subscribes to, R_1 is the RMI marshalling overhead per sensor, and R_2 is the RMI marshalling overhead per RMI call - generally it would be expected that R_1 is small compared to R_2. Finally it should be noted that because of the multiplexing in the wind farm service the number of subscribing clients is not expected to be a factor in any of these functions.

F.2 Measured Packet Numbers and Sizes

The number of network packets and the size of these was found through a combination of raw network analysis between the farm server and a turbine node, and by enabling the RMI transport level debug logging as described in [RMI Logging]. Based on this the actual flow of network packets was found to be as shown in figure 44 on the following page.

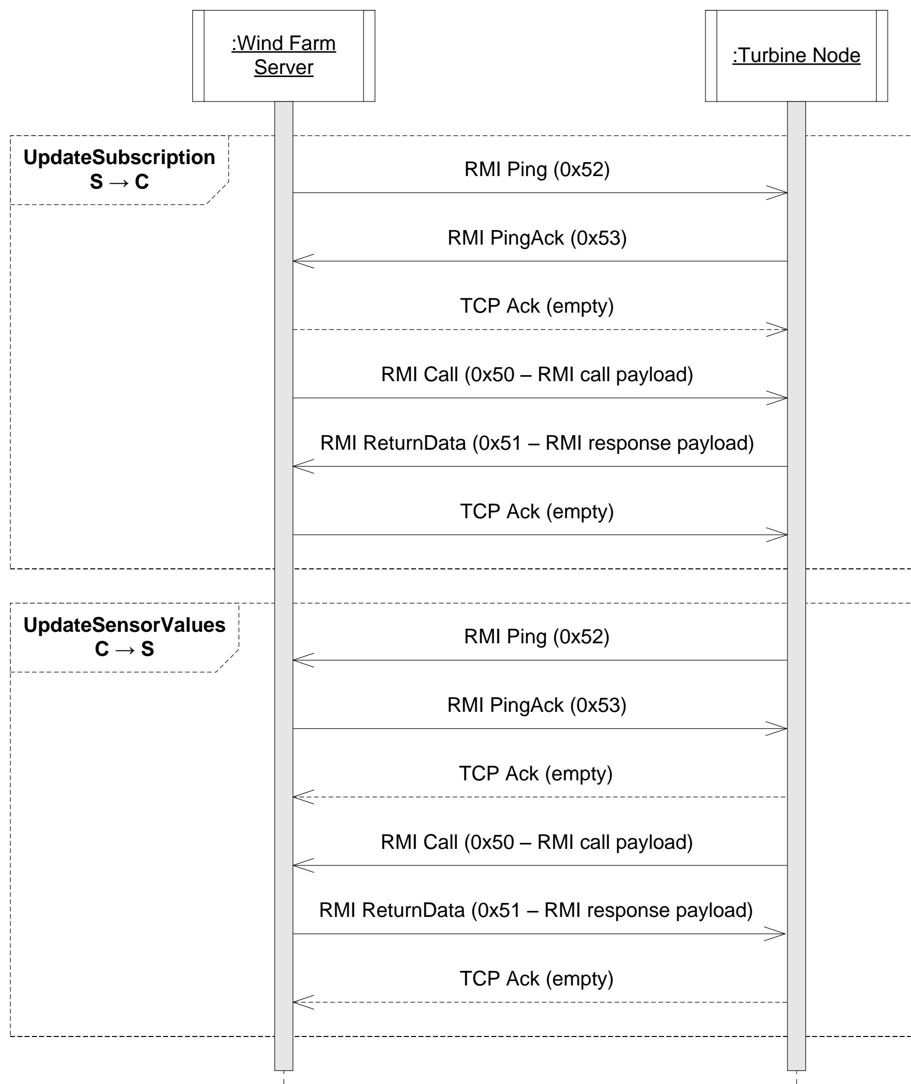


Figure 44: Network packet flow for RMI method calls.

The packet payload were analysed based on the information specified in [RMI Specification, ch.10.2]. Before the packet flow shown in the figure there are of course the UDP discovery packets, as well as some initial RMI packets. The initial RMI packets includes some JNDI lookup packets, as well as packets that initializes the RMI protocol to using the so called stream protocol (RMI message type 0x4b[21]). Basically this is the standard protocol used when a TCP stream connection can be established between a RMI client and server. Two other protocol exists SingleOpProtocol (type 0x4c) for usage when RMI is embedded in HTTP messages, and MultiplexProtocol (type 0x4d) for usage in some special scenarios (e.g. in applets). For further details on these different protocol types and their characteristics, the [RMI Specification, ch.10.2 and

[21]The first byte of a RMI message specifies the RMI message type.

ch.10.6] should be consulted.

From the figure it can be seen that for each RMI call, several TCP packets are exchanged. Before any call a *Ping*-message[22] (type 0x52), and a *PingAck* (type 0x53) are exchanged, followed by an empty TCP ack message. Only after this, the actual call is marshaled and send in a *Call*-message (type 0x50), replied to by a a *ReturnData*-message (type 0x51), followed by a standard empty TCP ack message.

Scrutinizing the headers of the exchanged TCP packets it can be seen that the so called TCP *Push* bit is set on all packets except the empty TCP ack messages. The ack messages could therefore in theory be piggybacked on other packets send between the relevant ports, but in the observed system this did not seem to be the case at any time. So using the postulated relation for the packet count $y = Mx + R_c$, M was measured to $(4 + 2)$ packets per RMI call, i.e. 4 individual packets, and 2 that in theory could be piggybacked. With respect to measuring R_c only a small number of messages related to ongoing RMI housekeeping was seen. As expected they seem to be related to the distributed garbage collection algorithm. An algorithm which is a simple reference counting GC algorithm based upon taking leases (see [Grosso, 2002, ch.16] for extensive in depth treatment of the actual working of this lease mechanism in Sun's standard RMI implementation). The number of these messages were insignificant compared to the traffic actively generated by the application. Basically there is just one RMI call (consisting of the usual $(4 + 2)$ TCP packets) to the java.rmi.dgc.DGC.dirty(..) method before the client first issues a call to an RMI object, and according to the documentation the default lease period is 10 minutes[23].

Previously in chapter 2.2.1 on page 14 it was described how the *broadcast client* on all turbine nodes periodically calls the farm server to detect if it becomes unavailable, so that the turbine node can enter reconnect mode if it happens. This call takes place once every second. As this is the same interval as the sensor value callbacks happens with, this effectively doubles the packet count Y related to RMI messages initiated from the turbine nodes to the farm server. But since the same one second interval is used for the two types of messages, it would be obvious to attempt to optimize on this, e.g. by disabling the broadcast client keep alive messages whenever at least one sensor is subscribed to at the turbine node. In reality there does not seem to be any bottleneck at this point though.

With respect to the packet sizes, this depends heavily on the actual data structures passed by value in the different RMI calls. The actual measured sizes for the different message types with 1 client making a subscription for respectively 1, 10 and 100 different sensors on the same turbine can be seen in table 8.

As expected the size of the *broadcast client* keep alive messages (labelled *isRegistered*), as well as the RMI garbage collection lease messages are independent of the number of subscribed sensors. Using the postulated relation for the packet sizes $Z = R_1 s + R_2$, the constants R_1 and R_2 can with trivial isolation of s using the measured values for Z for the request messages be calculated as

[22]Not to be confused with the standard network ICMP ping. The ping-message described here is a standard TCP packet, but the message type is called *Ping* in RMI terms.

[23]The lease period can be adjusted with the system property 'java.rmi.dgc.leaseValue'

# Sensors / RMI Call	1 sensor	10 sensors	100 sensors
updateSubscription (S → C)	Req: 816 B Resp: 22 B	Req: 1131 B Resp: 22 B	Req: 4371 B Resp: 22 B
updateSensorValues (C → S)	Req: 528 B Resp: 22 B	Req: 969 B Resp: 22 B	Req: 5469 B Resp: 22 B
isRegistered (C → S)	Req: 467 B Resp: 23 B	Req: 467 B Resp: 23 B	Req: 467 B Resp: 23 B
DGC.dirty	Req: 451 B Resp: 287 B	Req: 451 B Resp: 287 B	Req: 451 B Resp: 287 B

Table 8: Size in bytes of TCP payload for RMI messages in the baseline system.
C - > S: Request goes from wind turbine node to wind farm server.
S -> C: Request goes from wind farm server to wind turbine node.

shown in table 9[24]

RMI Call	R_1	R_2
UpdateSubscription	36 B	781 B
UpdateSensorValues	50 B	479 B

Table 9: Size in bytes for R_1 and R_2 in the relation $Z = R_1 s + R_2$

The differences in the values across the two different methods makes good sense as it depends on the actual parameters the methods takes as input. A verification of the serialized format of the relevant Java objects would without doubt verify these numbers, but this is out of scope of this thesis, and such an analysis should be pretty forward for any experienced Java developer, and although some exciting information about the optimizations and patterns used in the Java serialization mechanism could probably be learned by such an exercise, no further exploration of this will be done here[25].

Finally it was also verified whether the claim that the network traffic (in packet count and/or packet sizes) between the farm server and the turbine nodes was not influenced by the number of concurrent clients. This was found to be correct only as long as the clients subscribes to the exact same sensors. In the scenario where the clients sensor subscriptions are only partly overlapping sets, additional UpdateSubscription calls takes place every time a client is either the first to subscribe on a specific sensor, or the last to unsubscribe from a specific sensor. But with respect to the notification messages from the turbine nodes to the farm server the claim is correct in all scenarios as the multiplexing in the farm server multiplies the single notification to all the subscribing clients.

The raw network capture files for the above analysis, are found in form of pcap files in the source archive for this thesis in the rmi-analysis folder. One tool that will make detailed analysis of the raw data possible is [Wireshark]. An example of some of the essential information used in the analysis is depicted on

[24]The observant reader checking the calculations will find that the values of R_1 are not exact. Depending of which 2 of the 3 equations are used for isolating R_1 the result will be either 35 or 36 bytes for the UpdateSubscription call and 49 or 50 for the UpdateSensorValues call.

[25]The interested reader can find in depth coverage of the Java serialization format in [Halloway, 2002, ch.4].

figure 45. It should be noted that manual analysis of the data is cumbersome and time consuming. In the situation where the idea of doing detailed network packet analysis for all the prototypes would have been pursued, some scripts for automating the analysis tasks should probably have been developed, just as they were for the timing measurements.

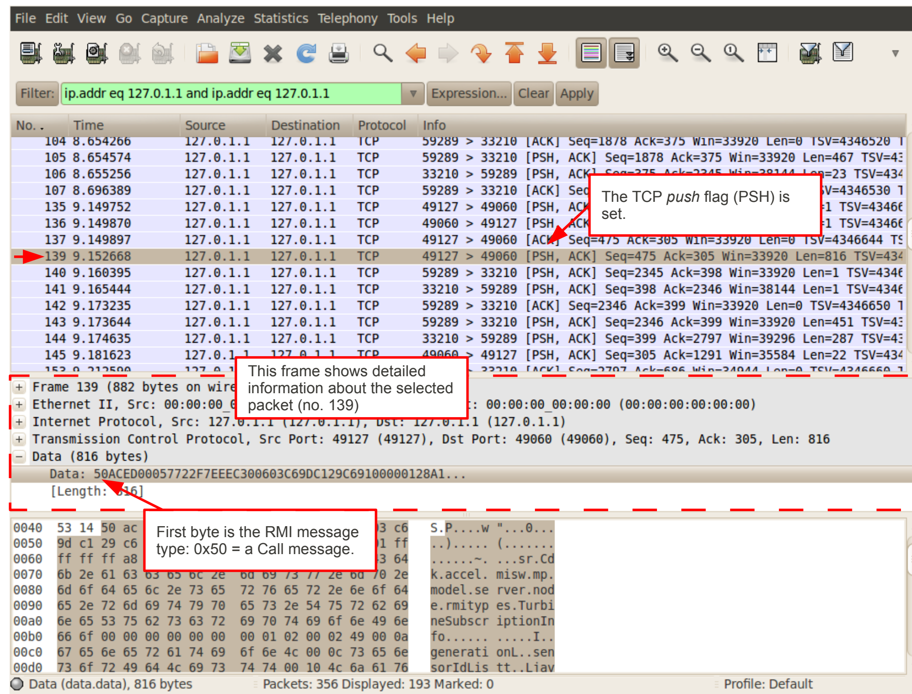


Figure 45: Wireshark analysis of the network capture files.

G Test Report for the Hazelcast Based Prototype

This appendix shows detailed measurements for 100 test runs of the Hazelcast based prototype and contains a detailed analysis of the problems that was found in the prototype.

Due to the large size of this appendix with 100 graphs, it is found in a separate document:

master2011-MA-appendix-G.pdf

H Test Report for the End-to-End Based Prototype

This appendix shows detailed measurements for 100 test runs of the end-to-end based prototype.

Due to the large size of this appendix with 100 graphs, it is found in a separate document:

master2011-MA-appendix-H.pdf

References

[Amazon EC2] Amazon Web Services. `https://aws.amazon.com/` (accessed June 5., 2011).

[Barbacci et al., 2003] Barbacci, M.R., Ellison, R., Lattanze , A.J., Stafford, J.A., Weinstock, C.B. and Wood, W.G. (2003). *Quality Attribute Workshops (QAWs)*, Third Edition, Carnegie Mellon, Software Engineering Institute, CMU/SEI-2003-TR-016, ESC-TR-2003-016.

[Bardram et al., 2004] Bardram, J. E., Christensen, H.B., and Hansen, K.M. (2004). *Architectural prototyping: An approach for grounding architectural design and learning.* In Proceedings of the 4th Working IEEE/IFIP Conference on Software Architecture, pages 15-24.

[Bass et al., 2003] Bass, L., Clements, P., and Kazman, R. (2003). *Software Architecture in Practice*, 2nd edition, Addison-Wesley, 2003. ISBN: 0-321-15495-9.

[Birman, 2005] Birman, K.P. (2005). *Reliable Distributed Systems.* Springer, 2005. ISBN: 0-387-21509-3.

[Burns & Wellings, 2001] Burns, A. and Wellings, A. (2001). *Real-Time Systems and Programming Languages*, 3rd edition, Addison-Wesley, 2001. ISBN: 0-201-72988-1.

[Christensen et al., 2007] Christensen, H., Corry, A., and Hansen, K. (2007). *An Approach to Software Architecture Description Using UML, Revision 2.0*, Technical report, Department of Computer Science, University of Aarhus, June, 2007.

[Clements et al., 2003] Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., and Stafford, J., editors (2003). *Documenting Software Architectures: Views and Beyond*, Addison-Wesley, 2003. ISBN: 0-201-70372-6.

[Coulouris et al., 2005] Coulouris, G., Dollimore, J., and Kindberg, T. (2005). *Distributed Systems, Concepts and Design*, fourth edition, Addison-Wesley, 2005. ISBN: 0-321-26354-5.

[Cristian, 1993] Cristian, F. (1993). *Understanding Fault-Tolerant Distributed Systems*, Computer Science and Engineering, University of California, San Diego, May 25., 1993.

[Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995). *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995. ISBN: 0-201-63361-2.

[GNU R] *GNU R - The R Project for Statistical Computing, version 2.10.1-2.* `http://www.r-project.org/` (accessed June 5., 2011).

[Goetz et al., 2006] Goetz, B. with Peierls, T., Bloch, J., Bowbeer, J., Holmes, D. and Lea, D. (2006). *Java Concurrency in Practice*, Addison-Wesley, 2006. ISBN: 0-321-34960-1.

[Gradle] *Gradle Build Tool, version 0.9.2.* `http://www.gradle.org/` (accessed June 5., 2011).

[Grosso, 2002] Grosso, W. (2002). *Java RMI*, O'Reilly, 2002. ISBN: 1-56592-452-5.

[Halloway, 2002] Halloway, S.D. (2002). *Component Development for the Java Platform*, Addison-Wesley, 2002. ISBN: 0-201-75306-5.

[Hazelcast] *The Hazelcast Project, version 1.9.3* (released May 30., 2011). `http://www.hazelcast.com` (accessed June 5., 2011).

[Hazelcast, 2011] *Hazelcast Documentation* (2011). `http://www.hazelcast.com/documentation.jsp` (accessed June 5., 2011).

[Jini] *Jini.org.* `http://www.jini.org` (accessed June 5., 2011).

[JSR-133, 2004] *JSR 133: Java Memory Model and Thread Specification Revision*, August 24, 2004. Available from `http://www.cs.umd.edu/~pugh/java/memoryModel/jsr133.pdf` (accessed June 5., 2011).

[Lamport, 1978] Lamport, L. (1978). *Time, Clocks and the Ordering of Events in a Distributed System*, in Communications of the ACM, Vol.21, No.7, July 1978, pages 558-565.

[Lamport et al., 1982] Lamport, L., Shostak, R. and Pease, M. (1982). *The Byzantine Generals Problem*, In ACM Transactions on Programming Languages and Systems, Vol.4, No.3, July 1982, pages 382-401.

[Mahemoff, 2006] Mahemoff, M. (2006). *Ajax Design Patterns*, O'Reilly, 2006. ISBN: 0-596-10180-5.

[McAllister & Vouk, 1996] McAllister, D.F. and Vouk, M.A. (1996). *Fault-Tolerant Software Reliability Engineering*, in Handbook of Software Reliability Engineering, pages 567-614, Lyu, M.R. (editor), McGraw-Hill, 1996. ISBN: 0-07-039400-8.

[Mosberger, 1993] Mosberger, D. (1993). *Memory Consistency Models*, TR 93/11, Department of Computer Science, The University of Arizona.

[Nygard, 2007] Nygard, M.T. (2007). *Release It! Design and Deploy Production-Ready Software*, Pragmatic Bookshelf, 2007. ISBN-10: 0-9787392-1-3.

[OPC DA, 2003] *OPC Data Data Access Custom Interface Standard*, version 3.00, OPC Foundation, March 4, 2003.

[RFC 3724] IETF RFC 3724: *The Rise of the Middle and the Future of End-to-End: Reflections on the Evolution of the Internet Architecture*, IETF, March 2004.

[Ristenpart et al., 2009] Ristenpart, T., Tromer, E., Shacham, H. and Savage, S. (2009). *Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds*, Proceedings of the 16th ACM conference on Computer and communications security (CCS '09), November 2009, pages 199-212.

[RMI Logging] *Java RMI Implementation Logging in JDK 6.* http://download.oracle.com/javase/6/docs/technotes/guides/rmi/logging.html (accessed June 5., 2011). Also available in the downloadable JDK 6 documentation.

[RMI Specification] *Java RMI Specification.* http://download.oracle.com/javase/6/docs/platform/rmi/spec/rmiTOC.html (accessed June 5., 2011). Also available in the downloadable JDK 6 documentation.

[Saltzer et al., 1984] Saltzer, J.H., Reed, D.P and Clark, D.D. (1984). *End-To-End Arguments In System Design*, ACM Transactions on Computer Systems, Vol.2, Issue 4, November 1984, pages 277-288.

[Sommerville, 2007] Sommerville, I. (2007). *Software Engineering*, eighth edition, Addison-Wesley, 2007. ISBN: 0-321-31379-8.

[Spolsky, 2004] Spolsky, J. (2004). *Joel on Software.* Apress, 2004. ISBN: 1-59059-389-8. Most of the articles in this essay collection are also freely avaliable at the authors website http://www.joelonsoftware.com/ (accessed June 5., 2011).

[Terracotta] *The Terracotta Scalability Platform, version 3.5.1* (released April 21., 2011). http://www.terracotta.org (accessed June 5., 2011).

[Terracotta, 2008] Terracotta Inc. (2008). *The Definitive Guide to Terracotta: Cluster the JVM for Spring, Hibernate and POJO Scalability*, Apress, 2008. ISBN: 978-1-59059-986-0.

[Terracotta, 2011] *Terracotta 3.5.0 Documentation* (2011). http://www.terracotta.org/documentation/product-documentation (accessed June 5., 2011)

[Waldo et al., 1994] Waldo, J., Wyant, G., Wollrath, A. and Kendall, S. (1994). *A Note on Distributed Computing*, Sun Microsystems Laboratories, Technical Report SMLI TR-94-29, November 1994.

[Wireshark] *Wireshark network packet analyzer.* http://www.wireshark.org/ (accessed June 5., 2011).