Francis Tam

Service Availability Standards for Carrier-Grade Platforms: Creation and Deployment in Mobile Networks

Tampere 2009
Francis Tam

Service Availability Standards for Carrier-Grade Platforms: Creation and Deployment in Mobile Networks

Thesis for the degree of Doctor of Technology to be presented with due permission for public examination and criticism in Tietotalo Building, Auditorium TB222, at Tampere University of Technology, on the 15th of May 2009, at 12 noon.
Abstract

The rapid development of the mobile network industry has raised considerably the expectations and requirements of the whole chain of stakeholders, from the end users through the mobile network operators and ultimately to the network equipment providers. A key expectation from an end user is service availability, which is a perception that services are continuously operational even in the presence of failures in the mobile network.

Network equipment providers have been using carrier-grade platforms to provide various support functions including high availability as reusable assets for products creation. The term carrier-grade refers to a class of systems used in public telecommunications network that deliver up to five nines or six nines (99.999% or 99.9999%) availability. The convergence of communications and information technology in the industry has led to more competition and pressure to reduce development efforts. By creating a service availability standard, Commercial Off-The-Shelf (COTS) software can be bought and integrated into a carrier-grade platform, enabling a company to focus on the core business and concentrate the resource investment onto new innovations.

This thesis explores the kind of service availability support needed for a carrier-grade platform and captures the essential ones for standardisation. It also aims at investigating the impact of using standards-based service availability software in the platform on the overall product development life cycle.

The main result of this research is reflected in having the proposed solutions of availability management and online software upgrade standardised by the Service Availability Forum. All the current network equipment providers have effectively endorsed them as the service availability standards in the mobile network industry. It has also created a set of robustness test suites and experimented on three standards-compliant implementations.

The thesis has contributed to the development of concepts, programming and administrative interfaces, and mechanisms related to supporting availability management and online software upgrade in a carrier-grade platform. It has also contributed to a testing methodology that addresses the often neglected robustness factor in the selection process for COTS availability management middleware.
Preface

The research work presented in this thesis has been carried out in the project “High Availability Services: Standardisation and Technology Investigation” during 2001-2006. The project was funded by Strategy and Technology, Nokia Networks (now part of Nokia Siemens Networks) and was conducted in Nokia Research Center. The objective was to support the company’s standardisation effort in the Service Availability Forum and contribute to a consistent carrier-grade base platform architecture for Nokia Networks’ business. I would like to thank Timo Jokiaho for having the confidence in me to explore new areas year after year. His open and direct manner of communicating and giving feedback were among the key factors to achieving these results. I am grateful to many Nokia colleagues who have worked on related projects for their co-operation. The granting of a leave of absence to prepare this thesis by Nokia Research Center’s management is also acknowledged.

This thesis would not have been a reality if it was not for Professor Tommi Mikkonen, who was brave enough to agree to be my supervisor after only a brief introduction at a conference. His guidance, incredibly quick turnaround when reviewing the various drafts, and consistently providing me with generous comments and suggestions have transformed a demanding writing period into an enjoyable experience. He has also kindly made financial support available at a critical stage of the thesis’ preparation.

I am thankful to Professor Hannu-Matti Järvinen, my second supervisor, for continuously giving feedback, advice, and suggestions throughout the study period; Professor Kai Koskimies for his thorough review and thoughtful recommendations; Professors Sascha Romanovsky and Jari Porras, the preliminary examiners, for their constructive suggestions that have led to various improvements in this thesis.

I have had the privilege of working with so many talented and experienced representatives from other member companies at the Service Availability Forum. Co-chairing the Software Management specification development group with Dr Maria Toeroe was one of the high points. Dr Fred Herrmann was an inspiration during the joint submission of our companies’ initial proposal that led to the first release of the Availability Management Framework. Collaborating with Zoltán Micskei and Professor István Majzik of Budapest University of Technology and Economics in Hungary to experiment with robustness testing techniques for high availability middleware was a rewarding adventure.

I am indebted to many colleagues in the dependability community, especially at conferences, workshops and seminars, for listening, questioning, arguing, and most importantly
showing me that there are always alternatives. I remain optimistic about better solutions in an otherwise pessimistic field of study – which suggests that computers could actually fail!

Two special people played pivotal roles in starting me off with the study in the first place. Dr Olli Karonen, a former manager of mine, planted the “compilation of papers” route idea into my head. His positive and subtle encouragement without any hint of pressure is most appreciated. Riikka Kallio, my better half, explained what the doctorate meant for us in a plain, succinct and yet powerful language. She deserves a medal for her unreserved support throughout the thesis preparation period and putting up with my long evenings and weekends of writing.

Francis Tam
Helsinki, February 2009.
## Contents

Abstract ........................................................................................................................................ iii

Preface ........................................................................................................................................ v

Contents ....................................................................................................................................... vii

List of Figures .............................................................................................................................. ix

List of Tables ............................................................................................................................... xi

List of Abbreviations .................................................................................................................. xiii

List of Included Publications ..................................................................................................... xv

1 Introduction ................................................................................................................................ 1
   1.1 Background ...................................................................................................................... 1
   1.2 Motivation ..................................................................................................................... 4
      1.2.1 Need for an Open Standard ..................................................................................... 4
      1.2.2 Minimising Service Outage ................................................................................... 5
      1.2.3 Buying the Right Software .................................................................................... 5
   1.3 Thesis Scope and Research Questions ............................................................................. 6
   1.4 Research Method .............................................................................................................. 7
   1.5 Contributions .................................................................................................................. 8
   1.6 Organisation ..................................................................................................................... 9

2 Carrier-Grade Platforms in a Mobile Network ......................................................................... 11
   2.1 A Mobile Network Primer ............................................................................................. 11
   2.2 Mobile Network Stakeholders ....................................................................................... 12
   2.3 Drawing the Software Product Line Parallel .................................................................. 14
   2.4 Support for Service Availability .................................................................................... 16
      2.4.1 Replication ............................................................................................................ 16
      2.4.2 Availability Management ...................................................................................... 17
      2.4.3 Online Software Upgrade ...................................................................................... 19
   2.5 The Role and Forms of Standards .................................................................................. 21
   2.6 Summary ........................................................................................................................ 23

3 Standards-Based Service Availability Middleware .................................................................. 25
   3.1 Why a Service Availability Standard? ........................................................................... 25
   3.2 Building Blocks .............................................................................................................. 26
   3.3 Service Availability Forum Middleware ......................................................................... 27
   3.4 Application Interface Specification ................................................................................. 28
   3.5 Hardware Platform Interface ......................................................................................... 29
   3.6 Creating a Service Availability Forum Specification .................................................... 32
   3.7 Summary ........................................................................................................................ 33

4 Availability Management Framework ..................................................................................... 35
   4.1 Overview ....................................................................................................................... 35
      4.1.1 Basic Model ............................................................................................................ 35
      4.1.2 Component States ................................................................................................. 38
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1.3</td>
<td>Service Group Redundancy Model</td>
<td>39</td>
</tr>
<tr>
<td>4.1.4</td>
<td>Component Capability Model</td>
<td>42</td>
</tr>
<tr>
<td>4.1.5</td>
<td>Programming and Administrative Interfaces</td>
<td>43</td>
</tr>
<tr>
<td>4.2</td>
<td>Architecting an Example GGSN</td>
<td>44</td>
</tr>
<tr>
<td>4.3</td>
<td>Additional Validation</td>
<td>47</td>
</tr>
<tr>
<td>4.4</td>
<td>Related Work</td>
<td>50</td>
</tr>
<tr>
<td>4.5</td>
<td>Summary</td>
<td>53</td>
</tr>
<tr>
<td>5</td>
<td>Software Management Framework</td>
<td>55</td>
</tr>
<tr>
<td>5.1</td>
<td>Overview</td>
<td>55</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Upgrade Campaign</td>
<td>56</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Upgrade Campaign Specification</td>
<td>57</td>
</tr>
<tr>
<td>5.1.3</td>
<td>Failure Handling</td>
<td>59</td>
</tr>
<tr>
<td>5.1.4</td>
<td>State Models</td>
<td>61</td>
</tr>
<tr>
<td>5.1.5</td>
<td>Programming and Administrative Interfaces</td>
<td>62</td>
</tr>
<tr>
<td>5.2</td>
<td>Upgrading the Example GGSN</td>
<td>63</td>
</tr>
<tr>
<td>5.3</td>
<td>Additional Validation</td>
<td>65</td>
</tr>
<tr>
<td>5.4</td>
<td>Related Work</td>
<td>67</td>
</tr>
<tr>
<td>5.5</td>
<td>Summary</td>
<td>71</td>
</tr>
<tr>
<td>6</td>
<td>Selecting a Standards-Based Availability Management Middleware</td>
<td>73</td>
</tr>
<tr>
<td>6.1</td>
<td>Motivation</td>
<td>73</td>
</tr>
<tr>
<td>6.2</td>
<td>Conformance Testing</td>
<td>75</td>
</tr>
<tr>
<td>6.2.1</td>
<td>The IEEE 2003-1997 Approach</td>
<td>75</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Experimental Results and Evaluation</td>
<td>77</td>
</tr>
<tr>
<td>6.3</td>
<td>Robustness Testing</td>
<td>80</td>
</tr>
<tr>
<td>6.3.1</td>
<td>Adaptation to AMF</td>
<td>80</td>
</tr>
<tr>
<td>6.3.2</td>
<td>Experimental Results and Evaluation</td>
<td>82</td>
</tr>
<tr>
<td>6.3.3</td>
<td>Extensions</td>
<td>84</td>
</tr>
<tr>
<td>6.4</td>
<td>Related Work</td>
<td>86</td>
</tr>
<tr>
<td>6.5</td>
<td>Summary</td>
<td>89</td>
</tr>
<tr>
<td>7</td>
<td>Introduction to the Included Publications</td>
<td>91</td>
</tr>
<tr>
<td>7.1</td>
<td>On the Development of Standards Based Carrier Grade Platforms</td>
<td>91</td>
</tr>
<tr>
<td>7.2</td>
<td>Fault Tolerant CORBA: A Telecommunications Perspective</td>
<td>92</td>
</tr>
<tr>
<td>7.3</td>
<td>On the Development of an Open Standard for Highly Available Telecommunication Infrastructure Systems</td>
<td>92</td>
</tr>
<tr>
<td>7.4</td>
<td>The Emerging SAF Software Management Framework</td>
<td>93</td>
</tr>
<tr>
<td>7.5</td>
<td>First Experience of Conformance Testing an Application Interface Specification Implementation</td>
<td>93</td>
</tr>
<tr>
<td>7.6</td>
<td>Robustness Testing Techniques for High Availability Middleware Solutions</td>
<td>94</td>
</tr>
<tr>
<td>7.7</td>
<td>Comparing Robustness of AIS-Based Middleware Implementations</td>
<td>94</td>
</tr>
<tr>
<td>7.8</td>
<td>Standards Contributions</td>
<td>95</td>
</tr>
<tr>
<td>7.8.1</td>
<td>Availability Management Framework</td>
<td>95</td>
</tr>
<tr>
<td>7.8.2</td>
<td>Software Management Framework</td>
<td>95</td>
</tr>
<tr>
<td>8</td>
<td>Conclusions</td>
<td>97</td>
</tr>
<tr>
<td>8.1</td>
<td>Analysis of the Overall Approach</td>
<td>97</td>
</tr>
<tr>
<td>8.2</td>
<td>Thesis Questions Revisited</td>
<td>98</td>
</tr>
<tr>
<td>8.3</td>
<td>Future Work</td>
<td>100</td>
</tr>
<tr>
<td>8.4</td>
<td>Concluding Remarks</td>
<td>101</td>
</tr>
</tbody>
</table>

References | 103

Included Publications | 111
List of Figures

Figure 1.1  Context of a carrier-grade platform ................................................................. 2
Figure 2.1  A simplified Public Land Mobile Network ....................................................... 11
Figure 2.2  Expectations and requirements ....................................................................... 13
Figure 2.3  Network element architecture ...................................................................... 15
Figure 2.4  Redundancy models ...................................................................................... 18
Figure 2.5  Rolling upgrade ........................................................................................... 19
Figure 2.6  Split-mode upgrade .................................................................................... 20
Figure 2.7  Forms of interaction .................................................................................... 22
Figure 3.1  Service Availability Forum middleware .......................................................... 27
Figure 3.2  HPI conceptual model .................................................................................. 31
Figure 3.3  Specification development process ............................................................... 32
Figure 4.1  Logical entities illustration .......................................................................... 37
Figure 4.2  State diagram of the HA state for a CSI ....................................................... 39
Figure 4.3  2N redundancy model ................................................................................ 40
Figure 4.4  N+M redundancy model ............................................................................ 41
Figure 4.5  Logical architecture of GPRS ...................................................................... 45
Figure 4.6  High level design of example GGSN ............................................................ 46
Figure 4.7  Example GGSN in 2N ................................................................................ 47
Figure 5.1  Upgrade campaign activities ....................................................................... 57
Figure 5.2  Upgrade campaign specification template .................................................. 58
Figure 5.3  Upgrade campaign activities with failure handling ..................................... 60
Figure 5.4  Upgrade campaign finite state machines ..................................................... 62
Figure 5.5  A pseudo upgrade campaign specification .................................................. 65
Figure 6.1  Network element life cycle ......................................................................... 73
Figure 6.2  Basic model for conformance assessment .................................................... 76
Figure 6.3 Get component capability model function................................................................. 78
Figure 6.4 Template-Based Type-Specific Test Generator......................................................... 82
Figure 6.5 Vision of dependability benchmarking..................................................................... 86
List of Tables

Table 3.1 AIS services ............................................................................................................... 29
Table 3.2 HPI basic concepts ..................................................................................................... 30
Table 3.3 Management instruments ........................................................................................... 30
Table 3.4 Management capabilities............................................................................................ 31
Table 4.1 AMF logical entities................................................................................................ ... 36
Table 4.2 Underlying states of a component.............................................................................. 38
Table 4.3 Component capability models.................................................................................... 42
Table 4.4 Component capability model and service group redundancy model mapping......... 43
Table 4.5 Availability Management Framework API ................................................................ 43
Table 4.6 Availability Management Framework Administrative API ....................................... 44
Table 5.1 Concepts associated with an upgrade campaign ........................................................ 56
Table 5.2 Software Management Framework API..................................................................... 63
Table 5.3 Software Management Framework Administrative API............................................ 63
Table 6.1 IEEE 2003-1997 definitions....................................................................................... 76
Table 6.2 Example assertions.................................................................................................. ... 78
Table 6.3 Analysis of exhaustive and thorough testing.............................................................. 79
Table 6.4 Number of lines of source codes ................................................................................ 83
Table 6.5 Number of faults by functions .................................................................................... 83
Table 6.6 Results of template-based type-specific testing ......................................................... 84
# List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3G</td>
<td>Third Generation</td>
</tr>
<tr>
<td>AIS</td>
<td>Application Interface Specification</td>
</tr>
<tr>
<td>AMF</td>
<td>Availability Management Framework</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>BSS</td>
<td>Base Station System</td>
</tr>
<tr>
<td>CGF</td>
<td>Charging Gateway Functionality</td>
</tr>
<tr>
<td>CORBA</td>
<td>Common Object Request Broker Architecture</td>
</tr>
<tr>
<td>COTS</td>
<td>Commercial Off-The-Shelf</td>
</tr>
<tr>
<td>CS</td>
<td>Circuit Switched</td>
</tr>
<tr>
<td>CSI</td>
<td>Component Service Instance</td>
</tr>
<tr>
<td>DIMI</td>
<td>Diagnostic Initiator Management Instrument</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processor</td>
</tr>
<tr>
<td>FSM</td>
<td>Finite State Machine</td>
</tr>
<tr>
<td>FUMI</td>
<td>Firmware Upgrade Management Instrument</td>
</tr>
<tr>
<td>GGSN</td>
<td>Gateway GPRS Support Node</td>
</tr>
<tr>
<td>GPRS</td>
<td>General Packet Radio Service</td>
</tr>
<tr>
<td>GSM</td>
<td>Global System for Mobile communications (Groupe Spécial Mobile)</td>
</tr>
<tr>
<td>HA</td>
<td>High Availability</td>
</tr>
<tr>
<td>HLR</td>
<td>Home Location Register</td>
</tr>
<tr>
<td>HPI</td>
<td>Hardware Platform Interface</td>
</tr>
<tr>
<td>HSS</td>
<td>Home Subscriber Server</td>
</tr>
<tr>
<td>IDL</td>
<td>Interface Definition Language</td>
</tr>
<tr>
<td>IDR</td>
<td>Inventory Data Repository</td>
</tr>
<tr>
<td>MS</td>
<td>Mobile Station</td>
</tr>
<tr>
<td>OMG</td>
<td>Object Management Group</td>
</tr>
<tr>
<td>ORB</td>
<td>Object Request Broker</td>
</tr>
<tr>
<td>PDN</td>
<td>Packet Data Network</td>
</tr>
<tr>
<td>PDP</td>
<td>Packet Data Protocol</td>
</tr>
<tr>
<td>PLMN</td>
<td>Public Land Mobile Network</td>
</tr>
<tr>
<td>PS</td>
<td>Packet Switched</td>
</tr>
<tr>
<td>PSTN</td>
<td>Public Switched Telephone Network</td>
</tr>
<tr>
<td>RFP</td>
<td>Request For Proposal</td>
</tr>
<tr>
<td>RPM</td>
<td>Redhat Package Manager</td>
</tr>
<tr>
<td>Acronym</td>
<td>Full Form</td>
</tr>
<tr>
<td>---------</td>
<td>-----------</td>
</tr>
<tr>
<td>SAF</td>
<td>Service Availability Forum</td>
</tr>
<tr>
<td>SGSN</td>
<td>Serving GPRS Support Node</td>
</tr>
<tr>
<td>SLA</td>
<td>Service Level Agreement</td>
</tr>
<tr>
<td>SMF</td>
<td>Software Management Framework</td>
</tr>
<tr>
<td>TBTS-TG</td>
<td>Template-Based Type-Specific Test Generator</td>
</tr>
<tr>
<td>TMI</td>
<td>Test Method Implementation</td>
</tr>
<tr>
<td>TMS</td>
<td>Test Method Specification</td>
</tr>
<tr>
<td>XDSA</td>
<td>Distributed Software Administration</td>
</tr>
<tr>
<td>XML</td>
<td>eXtensible Markup Language</td>
</tr>
<tr>
<td>XSL</td>
<td>eXtensible Stylesheet Language</td>
</tr>
<tr>
<td>UMTS</td>
<td>Universal Mobile Telecommunication System</td>
</tr>
</tbody>
</table>
List of Included Publications


The permissions of the copyright holders of the original publications to reprint them in this thesis are hereby acknowledged.
1 Introduction

A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.

– Leslie Lamport

The nice thing about standards is that you have so many to choose from.

– Andrew S. Tanenbaum

1.1 Background

The remarkable pace of telecommunications technology advancement have fuelled the rapid development of the mobile network industry. It took only a few decades to progress from the analogue system to the digital GSM [30] networks and onto the third generation (3G) [3] systems, where users can also use data services for sending pictures in multimedia messages, video conferencing or even watch television while on the move.

From a producer-consumer perspective, the mobile network industry can be viewed as a three-tier hierarchy. At the bottom are the network equipment providers, who sell network infrastructure devices as mobile networks building blocks to the mobile network operators. In the middle layer, mobile network operators run their businesses by providing services such as voice communications, text messaging and other data services on the mobile network to the end users. At the top of this hierarchy, end users subscribe to a mobile network operator’s plan and pay for the usage of the services. Because of this relationship, the network equipment providers have to satisfy a long list of requirements and meet the high expectations that have been passed downwards.

A key expectation is service availability, which is a perception that services are continuously provided to an end user even in the presence of failures in some underlying systems in the mobile network infrastructure. Otherwise subscribers would not even consider using a service, let alone paying for something that may or may not work. This user perception necessitates a combination of highly available network equipment with the capability to maintain uninterrupted user sessions to give the impression of continuous service.

Software is nowadays the major ingredient in network infrastructure devices similarly to many other industries. Software systems in telecommunications have grown in size and
complexity. A typical subsystem contains millions of lines of code, making development effort for software a key part of the total costs. While there are many ways of achieving cost reduction, an approach based on a software product line [46] lends itself naturally to network infrastructure devices development. By distinguishing between development for reuse in domain engineering and development with reuse in application engineering [89], a platform provides reusable assets as a basis for individual product creation.

To a certain extent, the network equipment providers have been practising a product line approach for a long time by having a common element known as a carrier-grade platform in their products. The term carrier-grade refers to a class of systems used in public telecommunications network that deliver up to five nines or six nines (99.999% or 99.9999%) availability. Its origin comes from the fact that telecommunications company that provides the public for hire with communications transmission services is known as a carrier [5]. The equipment associated with providing these highly available services has traditionally been dubbed “carrier-grade”.

Availability is a function of time and is defined as the probability that a system is operating correctly and is available to perform its functions at a particular instant of time. It is expressed as:

\[
\text{Availability} = \frac{t_{\text{op}}}{t_{\text{op}} + t_{\text{repair}}}
\]

where \(t_{\text{op}}\) is the time a system is operational and \(t_{\text{repair}}\) is the down time. The 99.999% availability allows for just five minutes 15 seconds of downtime in one year’s continuous operation.

Figure 1.1  Context of a carrier-grade platform
Figure 1.1 shows the context in which a carrier-grade platform is used. The main function of a carrier-grade platform is to provide all the commonly used services such as hardware abstraction, configuration, high availability, and software management to an application. Depending upon the application that runs on the carrier-grade platform, the result will be a product from Push-to-talk over Cellular [4] to Lawful Interception Gateway [2]. The scalability advantage comes from the wide selection of the underlying hardware. Depending upon the capacity and budget requirements of a site, the hardware could range from a single node server to highly scalable blade clusters. With this carrier-grade platform, the investment of effort can now be focused on the applications. The creation of a specific network infrastructure device product family is essentially a porting exercise of an application, albeit non-trivial, to a range of appropriate deployment hardware. Extending this use of common platforms to a wider range lends itself naturally to the principle of a software product line, where successful large-scale reuse has been reported in [89].

In spite of the apparent good match of utilising software product line’s notion of turning the carrier-grade platform into a common, reusable asset for many product groups, the up taking has not been as widespread as it should be. Very often, the number of carrier-grade platforms in a company is almost as high as the number of products. As the author has witnessed in one of the network equipment manufacturers, three common platform programmes were initiated during a five-year period, all of which failed to achieve the main goal of getting the proposed platform adopted by the respective product groups.

There were many observed reasons. The first was that such programmes tend to be too ambitious, taking on too many areas of reuse possibilities at any one time. The lesson learnt was to narrow down to a small number of high impact areas. The second perceived problem was down to the flaws in the proposed platform architecture, compromised primarily by the established product boundaries. The so-called common platform architecture was merely a collection of existing products strung together, reflecting more the organisational structure than the intended functionality. A complete redesign from scratch is one solution but this is easier said than done in the telecommunications industry, where legacy software continues to play an important role in new system deployment. Similar experiences shared by a number of network equipment providers have also been acknowledged, especially during informal discussions at standardisation meetings, conferences, seminars and the like.

Given the high profile of reporting in the news whenever mobile network operators fail to provide services, service availability has been widely recognised as a key common concept in the common platform architecture. For instance, the recently reported failures of Elisa in Finland [93] and Telia in Sweden [51][52] have affected one million and more than four million

---

1 Due to the sensitive nature of this topic from the commercial perspective, there are no documented references available.
subscribers respectively. Although subsequent attempts on common platforms have sharpened
the focus on service availability, which is a key and common characteristic across network
infrastructure devices, there is a need to satisfy the diverse availability requirements of
individual network infrastructure device applications.

1.2 Motivation

This section provides the motivations behind the three threads of investigation in this thesis.
They are the need for an open standard, minimising service outage, and buying the right
software.

1.2.1 Need for an Open Standard

Historically, a network equipment provider typically constructed their products from the
application all the way down to the hardware, which took a considerable amount of
development time and costs. One popular solution as witnessed in the mobile network industry
is to buy the constituent parts whenever possible, instead of building one’s own. The experience
in the late 1990 suggests that if a company were to continue with building everything in-house,
it would have needed to hire thousands of new developers each year [89] (in Chapter 13 Nokia
Networks).

Nowadays, network equipment providers have already replaced:

- custom-built processors by their commercially available counterparts,
- proprietary hardware by standards-based computing elements such as the PICMG’s
  AdvancedTCA [69], and
- in-house development of systems software by standard operating systems such as
  Carrier Grade Linux [53].

Due to the latest convergence of communications and information technology, the trend of
using more standardised components continues to move upwards to the high availability
functionality level where the development of service availability middleware as building blocks
for network infrastructure devices, systems, services and the like has apparently become
feasible.

Originally developed for supporting interoperability of applications under a distributed
architecture, middleware is a piece of software that connects a number of other software
together, enabling the resulting system to run on one or more machines across a network [10].
Middleware is positioned between the application software and potentially different operating
systems, and aims to reduce the overall complexity by providing a set of common
functionalities for the applications to interact with each other. The use of middleware in
database systems and transaction monitors have proved to be effective. The hypothesis is that a
service availability middleware can be equally successful in the construction of carrier-grade platforms. This suggests that a suitable candidate must capture the appropriate service availability functions and provide a mechanism for implementing the corresponding capabilities.

Adopting a standards-based approach in this buy-over-build tactic is considered to be a sound risk management strategy to avoid single-vendor lock-in. In addition, compatibility of products and subsystems delivered by different vendors can be ensured. This calls for an open standard of service availability specifically developed for the mobile network industry.

1.2.2 Minimising Service Outage

Introducing service availability capabilities into a carrier-grade platform entails more than just providing support for error detection, diagnostics and recovery to the applications in the form of availability management. As network infrastructure devices tend to have a long mission time in the field, support of online upgrade of the applications with no or minimum service outage must also be offered.

Although hardware technologies have advanced to a point where the replacement of physical components can be performed without powering off a system, changing software, whether it is for bug fixing or version upgrade, may lead to a potential stoppage due to the need of restarting or reloading of data in some cases. Currently, this kind of unavailability of service is usually covered by the scheduled downtime as stated in the Service Level Agreements (SLA) between the mobile network operators and network equipment providers. Due to the loss of revenue as a result of downtime, there is a growing trend for the SLA to reduce, and ultimately remove this kind of scheduled maintenance time periods altogether. In turn, this has led to the increasing pressure put on the network equipment providers to support online upgrade of applications as one of the priority features.

Online software upgrade is one of the most difficult issues to resolve since any solution must work well with the rest of the system, particularly the availability management mechanism that is already put in place in the carrier-grade platform. Moreover, there has not been any significant breakthrough in the field since the last reported event [21]. On the contrary, there have been numerous reports of service outage as a result of upgrade failures. For example, the latest blackout of stock trading for more than five and a half hours in the Nordic region [33] have caused considerable financial losses.

1.2.3 Buying the Right Software

Having an open standard for service availability is only half way to having a solution based on a standards-based approach of constructing carrier-grade platforms. From a buyer’s viewpoint, the way in which a compliant availability management software is chosen must be addressed.
Apart from conformance to a stated standard, robustness of a candidate product must be demonstrated that it copes well with stressful environmental conditions and exceptional input [36]. This stems from the current practice that when a product is considered from a pool of suppliers, robustness is usually overshadowed by performance. The “seems to work well” criterion is normally the most commonly used without any further consideration of the robustness of an implementation. In this context, the crashing or hanging of the third party availability management software will cause the whole network infrastructure device to fail as well.

Robustness testing [44] aims at increasing the understanding of the failure modes of Commercial Off-The-Shelf (COTS) software and is a promising approach. However, it has not been fully explored in the context of integrating into the software development life cycle, which requires the necessary methodologies and tool concepts be developed. This need is more pronounced when one considers that there are many network infrastructure device products using the carrier-grade platform, each of which is likely to have a slightly different robustness requirement on the selected third party availability management software.

### 1.3 Thesis Scope and Research Questions

This thesis explores different approaches of capturing the essence of service availability and its subsequent provisioning into a standards-based middleware for the mobile network domain. The scope is limited to the application support for service availability in a carrier-grade platform, and the potential impact of using a standards-based middleware on the overall product life cycle.

Since a carrier-grade platform is an enabler for facilitating and supporting the dependability of a network infrastructure device application, its fault model should not be mixed up with the fault assumptions in the resulting network infrastructure device, which is inherently application specific. The concepts and taxonomy of dependable and secure computing by Avizienis et al. [7] classifies faults that may affect a system during its life cycle in the development phase or use phase. There are three major partially overlapping groups of faults:

- **Interaction faults** which include all fault classes that occur during the use phase of a system.
- **Physical faults** which include all fault classes that concern hardware.
- **Development faults** which include all fault classes that may arise during the development phase of a system.

This thesis focuses on the capabilities needed in a carrier-grade platform to provide applications with the support for handling interaction faults. Since the use phase of a system involves the environment in which an application operates, the investigation will therefore
include the system administration related issues such as configuration, recovery and repair actions.

Although the treatment of physical faults is outside the scope of this thesis, there is an assumption that protective redundancy such as multiple processors, memory subsystems, communication paths and power supplies are used. This enables software and its associated data to be replicated on redundant hardware, and are made available even when some of the underlying hardware fail.

In the context of constructing a carrier-grade platform, the traditional way of integrating the product creation process with advanced methods and tools related to faults prevention, removal and forecasting [29] is normally used to deal with development faults. This is a subject on its own and its study is excluded from the thesis.

While security in general is placed under interaction faults in the environment during the use phase of a system, in practice it is adequate to use support such as transport level security protocols, authentication and secure command line interpreters in a carrier-grade platform. This is because network infrastructure devices are connected by separate internal and external subnets according to the type of traffic that flows through. These subnets are protected by appropriate gateway or firewall devices whenever there is a need for traffic to be routed from one subnet to another, or from one network to another. The topic of security will not be considered further in the thesis.

The key challenges and specific research questions of the thesis are:

- What kinds of service availability capabilities are needed in a carrier-grade platform for network infrastructure devices?
- What types of constructs are worth standardising in a service availability middleware?
- How can online software upgrade with minimum or no service interruption be supported on a carrier-grade platform?
- What sorts of methodologies and tool concepts are needed to support a product selection process for a third-party, standards-based availability management software?
- To what extent can a standards-based service availability middleware be utilised to construct a carrier-grade platform for network infrastructure devices?

1.4 Research Method

Constructive Research [50] aims at developing novel solutions to theoretically and practically relevant problems. The research result is a construct which has been created, instead of being discovered. This has been widely used in computer science and software engineering research. In this context, a construct can be a theory, algorithm, software or framework for a specific problem. It is crucial that the research conclusions are objectively argued, and supported by the validation of the developed construct against some predefined criteria using qualitative or
quantitative methods. In addition, the evaluations must show that the constructs have practical or theoretical relevance.

In this thesis, the developed constructs are the Availability Management Framework, Software Management Framework, and the application of robustness testing in supporting the selection of a COTS standards-based availability management middleware.

Since both the Availability Management Framework and Software Management Framework have already been standardised by the Service Availability Forum, technical evaluation of these two Frameworks had therefore been carried out during the specifications’ creation by means of the rigorous standards development process as described in Section 3.6 on page 32. Therefore, this thesis does not contain the evaluation of the two Frameworks presented in the traditional manner found in other theses as such, instead, additional validation of these Frameworks by a case study are used to reinforce that the solutions are relevant and applicable in the mobile network industry.

The evaluation of robustness testing in supporting the COTS availability management middleware selection is conducted by experiments. The usefulness of the developed robustness testing suites is demonstrated by testing the hypothesis that the degree of tolerating exceptional input and stressful conditions in standards-based availability management middleware increases as the relative maturity of development grows.

1.5 Contributions

The thesis has contributed to the knowledge of constructing carrier-grade platforms using standards-based availability management middleware, enabling the building of network infrastructure device product families cost-effectively in the mobile network industry. It has also created an essential understanding of the impact this approach has on aspects of product development, in terms of the architecture, process, tools and programming environments.

The thesis has contributed to and formed part of two published standard specifications on service availability. As a technical representative of Nokia to the Service Availability Forum, the author has contributed to the Availability Management Framework in the first release of the Application Interface Specification [75], the requirements of online software upgrade and subsequently the Software Management Framework specification [78].

Specifically, the contributions are:

- A standardised set of logical entities, which represent hardware and software, and their relationships for the purpose of availability management by means of error detection and isolation, recovery, and repair.

- A standardised way of representing workload of software entities and a novel approach of deferring the decision of choosing a redundancy scheme until deployment time.
- A standardised set of concepts associated with online software upgrade and the necessary failure handling support for no, or minimum service interruption.
- A standardised structuring mechanism for expressing online software upgrade and an original approach to select the most appropriate upgrade method on a per-site basis.
- A testing methodology to address the often neglected robustness factor in the selection process for COTS availability management middleware and an adaptation of dependability benchmarking for introducing robustness test suites as a reusable asset in a software product line.

The author’s contributions are presented in Chapters 4, 5 and 6. A detailed account of the author’s contribution in included publications, the published standards, and the author’s role in the Service Availability Forum are in Chapter 7.

1.6 Organisation

The thesis consists of eight chapters and seven included publications. Chapter 2 elaborates on the context and approach of the thesis. Chapter 3 examines the high availability constructs that are worth standardising in a carrier-grade platform and presents Service Availability Forum’s middleware. Availability Management Framework is explained in Chapter 4, together with an evaluation on the availability management support against a case study. Chapter 5 describes the Software Management Framework and assesses how well it supports online software upgrade. Chapter 6 discusses the needs of conformance testing and robustness testing in a product selection process, and explains the techniques developed for measuring and comparing the robustness of COTS service availability middleware. The included publications are introduced in Chapter 7 with a summary of the author’s contributions, as well as the author’s role in the Service Availability Forum. Chapter 8 concludes with an analysis of the overall approach, assessment of how the research questions have been addressed, and a speculation about future developments.
2 Carrier-Grade Platforms in a Mobile Network

This chapter gives an introduction to the constituent components of a typical mobile network and explains the interrelationship among the stakeholders. It presents the motivation for following the software product line approach by network equipment providers. Issues concerning the reuse of availability management and online software upgrade support in a carrier-grade platform are articulated. The potential benefits of using standards-based components to implement these reusable assets are discussed.

2.1 A Mobile Network Primer

In a network infrastructure, logical entities known as network elements are used in the provisioning of a telecommunications service [26]. These network elements can be a facility or equipment, each of which is typically implemented by one or more physical devices distributed throughout the network.

A Public Land Mobile Network (PLMN) is established for the purpose of providing land mobile telecommunications services to the public. Figure 2.1 shows a simplified configuration of a PLMN architecture of the Universal Mobile Telecommunication System (UMTS) [3], illustrating the essential network elements.
An Access Network is the point at which mobile devices gain entry to the mobile network and obtain their subscribed communications services, whereas a Core Network performs all the necessary functions related to the management of calls including connection, quality of service and management of mobile devices moving from one location area to another.

A Core Network is further divided into Circuit Switched (CS) and Packet Switched (PS) domains. A CS domain provides a connection-oriented transportation of user traffic, in which dedicated network resources are allocated at the connection establishment and freed at the connection release. Voice communications traffic are usually carried through the CS domain. A PS domain arranges user information as independent concatenation of bits in the form of packets, each of which is routed through the domain independent from its previous one. Data for mobile services such as Multimedia Messaging, Short Message, Internet communications with emails and World Wide Web access on mobile phones are transported through the PS domain.

In a CS domain, the Mobile-services Switching Centre Server and the Circuit Switched Media Gateway Function network elements transport the control signalling and user data respectively between a mobile device and the Public Switched Telephone Network (PSTN) as shown.

In a PS domain, data packets are typically provided by a General Packet Radio Service (GPRS) network. Traffic associated with mobile data services are conveyed to a Packet Data Network (PDN) such as the Internet. The Serving GPRS Support Node (SGSN) and Gateway GPRS Support Node (GGSN) network elements store correspondingly the subscription and routing information for registered mobile devices that interact with a PDN.

Common to the two domains is the Home Subscriber Server (HSS) network element, which stores the unique identifications of mobile devices and users’ subscription-related information of the PLMN.

2.2 Mobile Network Stakeholders

It must be emphasised that the level of complexity in a mobile network is extremely high. Some major functional entities such as Charging, Short Message Service, IP Multimedia Subsystem, other types of access networks, additional interfaces for roaming etc. have been omitted from the above diagram for the sake of clarity. In spite of this complexity, the mobile network stakeholders form a relatively straightforward provider-consumer relationships among themselves in practice. They are:

- **End users** who subscribe to a mobile network operator’s plan for paid usage of the provided communications services.
- **Mobile network operators** who acquire the necessary radio spectrum licence from the government, build the mobile network by assembling the network elements bought from
network equipment providers and provide mobile communication services to end customers.

- **Network equipment providers** who sell network elements such as those described in Section 2.1 to the mobile network operators.

With this hierarchy of stakeholders’ roles, the network equipment providers need to tackle some of the most stringent challenges in the mobile network industry. As a result of the rapid developments in the mobile communications industry, Figure 2.2 portrays the never-ending expectations that funnel down from the end users and mobile network operators as requirements to the network equipment providers.

At the top layer are the end users with the typical expectation that *functional enhancements* are frequently introduced by the mobile network operators. Recent trends have indicated that these enhancements are released every few months. The provided services should also be *affordable* and *dependable*, otherwise subscribers would not even consider using the service, let alone paying for a service that may or may not work.

![Figure 2.2 Expectations and requirements](image)

The second layer represents the mobile network operators. The challenge for this group is to be able to respond to the diverse market needs in a timely manner. This group must have low implementation and operating costs in order to meet the affordability expectation from the end users. The underlying communications infrastructure equipment must have *high performance* to cater for the potentially large number of service subscribers and the *flexibility* for deploying new features very swiftly. *Scalability*, in terms of handling more computations after appropriate resources have been added, is an important requirement for coping with the increasing demand for a service after its introduction. Last, but not least, the network equipment must be highly
available. It must also support the notion of \textit{service availability} by enabling services to run continuously in spite of any failure in individual systems. Failing to maintain uninterrupted user sessions would have a direct impact on the revenue generation for this group.

At the bottom layer are the network equipment providers. Since their customers are the mobile network operators, they have to satisfy the long list of requirements and meet the high expectations that have been passed downwards. As if this is not enough, the fierce competition in this industry pushes this group to have an ever shorter \textit{time-to-market}, at lower \textit{development costs}, in order to win businesses.

Note that Figure 2.2 is used only as an illustration, in a non-exhaustive way, that these issues are interrelated and highlight the fact that the solutions to these challenges must have a delicate balance. The focus of the figure is limited to the service availability aspect. Indeed there are many other perspectives of viewing this three-tier hierarchy of expectations and requirements. As an example, simple user interface and free services will have impact on the mobile device and business model respectively. However, they are beyond the scope of this thesis.

2.3 \textbf{Drawing the Software Product Line Parallel}

While there are many ways of reducing development costs and effort, such as model-driven architecture [63] and automatic code generation [34][57], reusing software assets appears to be the most intuitive approach due to the size and complexity of telecommunications software. Software assets may include requirements, designs, architectures, test cases and ultimately the code. Interestingly, network equipment providers have been practising some form of reuse already by utilising a common carrier-grade platform across a number of network infrastructure devices as shown in Figure 1.1 on page 2, which is quite similar to the software product line approach.

A \textit{software product line} refers to the engineering techniques for producing a collection of similar software systems from a shared set of software assets [85]. The foundation of a software product line is based on the distinction of \textit{development for reuse} in domain engineering and \textit{development with reuse} in application engineering [89]. In domain engineering, it focuses on developing a set of shared, reusable assets that are needed for creating specific products. In application engineering, a new product is built by using as much reusable assets from domain engineering as possible.

The key to building a software product line to support a range of products is to separate what is expected to remain constant across all members in the product family and what is expected to vary [8] (Chapter 14 Software Product Lines), broadly known as \textit{commonality} and \textit{variability} respectively. A crucial activity is to identify from the reusable assets the appropriate \textit{variation points}, where the differences in the final systems are located. This must be supported
by appropriate mechanisms for realising the variation points. Configurable source code, compile
time configuration, and runtime configuration are some of the examples. The time during which
decisions are made to realise a variation point is known as binding time [45]. Examples of the
different binding times include source reuse time, development time, build time, install time,
startup time and runtime. This offers different levels of flexibility to application engineering.

Although a carrier-grade platform serves mainly as a means to increase the portability of
network infrastructure applications onto a range of hardware, as well as a stage facilitating
future extensions, it is nevertheless a shared, reusable asset in disguise. As a first step of
investigating its potential synergy with the software product line approach, the commonalities
and variabilities of a carrier-grade platform must be understood.

Figure 2.3 shows the typical architecture of a network element. The carrier-grade platform
is shown enclosed with dashed lines, which potentially consists of the shared, reusable assets
developed in domain engineering for a family of network element products.

![Network element architecture](image)

**Figure 2.3 Network element architecture**

At the top is the *application* that runs on the carrier-grade platform. The application
determines what the network element is.

At the bottom is *hardware*, which shows the typical redundant communication networks
used in deployment. For example, redundant interconnect such as Ethernet is used in loosely
coupled systems, while replicated backplane bus structure is utilised in closely coupled
implementations. The interconnected nodes typically contain hardware resources such as CPUs, Network Processors, Digital Signal Processors (DSP), disk storage and switches. Typically, nodes with general purpose CPUs are used to host applications to provide value added services, whereas nodes with Network Processors and DSPs are primarily used for providing routing, forwarding and connection to other equipment in the network.

In the middle is the carrier-grade platform, which is made of the operating system, high availability function, and carrier-grade extensions.

The operating system controls the hardware resources and provides an abstract view upon which application software can be built. The key benefit is that software above this level can be decoupled from the hardware.

The high availability function provides the applications with the support of service availability capabilities that include fault management, which involves the detection of faults and sending of notifications to the concerned parties for possible recovery. Load balancing and load sharing are typically considered as part of this category as most of these schemes tend to exploit the inherent redundant resources in the system during the error free periods of operation.

The carrier-grade extensions usually contain those functions that are typically used on the platform. These include general purpose functionality such as performance management, configuration management and directory services, and telecommunications specific ones such as signalling protocol stacks, alarm management, to name just a few. These functions are outside the scope of this thesis.

Although the service availability support in the high availability function is a key and common characteristic of network elements, reusing these capabilities systematically has been very limited in practice. There is a need to investigate whether reusing service availability support is applicable in the context of carrier-grade platforms.

2.4 Support for Service Availability

In this section, the relationship of this thesis’ investigation with the general research on replication is discussed. The necessary application support for availability management and online software upgrade in a carrier-grade platform are explained. It also discusses the variation points identification for these two application support.

2.4.1 Replication

Replication is a means of making copies of data or other resources in a system [18] (in chapter 11 Replication). It is a key to providing enhanced availability by ensuring that failures can be tolerated simply because there is an exact copy of the data or other resources elsewhere. This gives the appearance that services are uninterrupted even when there are failures in the
underlying system. The investigation of supporting service availability can therefore be placed under the umbrella of general research on replication.

It is typical to distinguish between active and passive replication in systems that replicate data or services. The former is performed by processing the same request at every replica, while the latter processes every single request on a single replica and its state is transferred to other replicas. This requires that the behaviour of a replica must be deterministic, that is, having a predictable outcome every time by producing the same results with the same input. It must also be ensured that the states of a replica are consistent.

In a distributed system such as a network element as shown in Figure 2.3 on page 15, communications among replicas involve messages being transferred over the interconnected hardware. The notion of a process group [11] has been created to simplify the development of reliable distributed software. However, due to the non-determinism of some underlying communications networks, such as the Ethernet, additional algorithms [25] are needed to preserve the total order of messages in broadcast and multicast. In some applications, they can trade performance with a weaker order semantics such as causal ordering using the virtual synchrony model [13]. Agreement on the group membership has to be achieved in order for the functioning replicas to communicate [20]. Various agreement protocols to reach agreement under different failure assumptions have been shown in [48].

All of the above issues related to replication are relevant and must be taken into consideration when arriving at the solutions of supporting service availability on a carrier-grade platform. In order to minimise service outage, the availability of an application must be carefully managed in the first place. In addition, upgrading an application without bringing the system down certainly increases service availability. These two topics form the much needed application support and will be examined further in the following sections.

2.4.2 Availability Management

By definition, the availability management capability in the platform is to deal with availability on behalf of the applications to achieve the overall service availability. This involves providing the applications with mechanisms for error detection, diagnoses and recovery. A well established means is to use protective redundancy, a notion that introduces additional, otherwise redundant resources into a system to cover for the failed elements and offer continuous service. The required and redundant resources must therefore be coordinated. A redundancy model can be thought of as a structural representation of these resources and how they behave should there be a failure.

Figure 2.4 shows two types of commonly used redundancy models in network elements. In (a) and (b) an active-standby redundancy model is shown, where the service S is provided by the currently active application A running on node X and the standby application on node Y. If
the currently active application fails, the standby application will take over to provide the
service, as shown in (b). For stateful services, the internal state information of the active
application must be provided to the standby application for a seamless failover. For stateless
services, no such synchronisation is needed and the failover is almost immediate. If the failed
node is not repaired, a further failure will cause service outage.

Figure 2.4 Redundancy models

The N+1 redundancy model is illustrated in (c) and (d) where there are three (N=3) active
applications providing services with one standby application. Depending upon whether service
S is stateful or stateless, the required synchronisation between one of the active applications and
the standby application varies accordingly. If node Y fails, as shown in (d), the previous standby
application will become active and starts providing services. If the failed node is not repaired,
any further failure will not be protected. As a consequence, the level of service will decrease
until the service eventually becomes unavailable when the last node fails. This model allows for
a gradual service degradation. For stateful services, this incurs a synchronisation traffic load
from the active applications to the standby application. The load is proportional to the number
of active applications during normal operation, hence it may impact the overall service response
time. For stateless services, no such synchronisation is needed.

As can be seen, different redundancy models have different resource requirements,
limitations, and impact on how quickly a new active application can take control and restore
normal operations. The selection of which model to use lies with the applications where they are
fully aware of the product requirements. The key issue for the platform is to offer a flexible way
for deploying different redundancy models. In this respect, a redundancy model should be
treated as yet another source of variability.
2.4.3 Online Software Upgrade

Since network elements tend to have a long mission time, support of online software upgrade of applications with no or minimum service outage is essential in the platform as a basic service. Online software upgrade interrelates to the protective redundancy of a system because it is the redundant resources that keep the system running while some parts of it are being replaced. Furthermore, a substantial amount of support is needed:

- Software image management is necessary for the distribution, installation and configuration of software for the upgrade.
- Version management is required to differentiate the current edition of the running software from the one that is intended for the upgrade.
- A method is needed for describing the current and planned deployment configurations.
- A mechanism to bring the system to the intended configuration must exist.
- The monitoring and control of the upgrade, together with measures for error recovery of the upgrade process are essential.
- The upgrade process needs to be automated as experience has shown that human mistakes account for a considerable number of cases of system crashes during an upgrade by using error-prone manual processes in large-scale systems.

Depending on the application requirements, how the upgrade is carried out must be considered. Figure 2.5 illustrates a rolling upgrade method in which each application is taken out of service before upgrading to the new version one by one. The initial state is depicted in (a)
where there are three interconnected nodes, X, Y and Z, each of which runs an instance of application A version 1 in order to provide service S. (b) shows that the first upgrade that has taken place on node X, where the instance of application A is now running version 2 to provide service S. Both (c) and (d) illustrate that this procedure continues on nodes Y and Z until all the instances of application A have been upgraded to version 2. Throughout the upgrade services are available, although the overall service capacity is reduced by one serving application at the time when an upgrade is being performed. However, the underlying assumption is that both version 1 and version 2 of application A are compatible. As such, they can co-exist without causing any inconsistency in the system.

Figure 2.6 shows another upgrade method known as split-mode, which divides a system roughly into two halves that are upgraded one after another. The initial state of this example is in (a) where there are four interconnected nodes, W, X, Y and Z, each of which runs an instance of an application A version 1 to provide service S1. The split is indicated by the dotted line in (b), where half of the applications on nodes W and X are taken out of service and upgraded to version 2 of application A. At this point, the overall serving capacity is reduced to what nodes Y and Z provide. At point (c), a switchover to the newly upgraded service S2 is carried out by stopping version 1 of applications A on nodes Y and Z, followed by activating version 2 of application A on nodes W and X. The overall serving capacity is still reduced by the amount that would have been provided by nodes Y and Z. As soon as the upgrade of the applications on these nodes are complete, full service capacity is restored as shown in (d). This technique is primarily applicable when the old and new applications are incompatible.
In practice, there are many other upgrade methods, including some custom schemes that are needed to meet the requirements of a specific site in terms of its capacity planning and service level agreement. Again, the choice of upgrade method lies with the applications where they are fully aware of a network element’s service outage and compatibility requirements. The key issue for the platform is to offer a flexible way for applying different upgrade methods. Similarly, it should be treated as a variation point.

2.5 The Role and Forms of Standards

It has been shown in the previous section that the application support of service availability in a carrier-grade platform can be considered as a shared, reusable asset from a software product line perspective. This section explores the potential of implementing these identified reusable assets using standardised components. It also considers the most suitable form of interaction to be standardised for such kinds of assets.

Apart from reducing the risks of getting locked in by a single vendor when it comes to buying standardised components, a key role played by standards is to divide a huge mobile network into smaller pieces with well-defined boundaries. As a result, an ecosystem is created with different suppliers contributing to the different parts of a mobile network. The standardisation process ensures that the stakeholders are involved in the development and agreement of the outcome, resulting in conforming products being compatible and interoperable across interface boundaries. This is particularly important as very often the business deals are big and involving many vendors, the only way to ensure that the system as a whole works is to have standardised interfaces.

Another benefit of leveraging standards is the reduction of the overall complexity. As the previously proprietary platform is standardised, systems can now be created by adding the business critical functions on top, enabling a company to move up the value chain. As this happens, a company can focus on the core business and concentrate the resource investment into new innovations, adding value to the company during the development process. As a result, it is conceivable to offer the resulting network infrastructure device application software as a service [9], meeting the needs of the emerging Mobile Virtual Network Operators in some markets. This highlights one of the many opportunities for companies branching out into new areas that were previously very difficult, if not impossible. On the contrary, failing to move away from proprietary solutions towards open standards may cause a company dearly. A telecommunications market research analysis report about Nortel’s recent trouble in [87] is a case in point.

In general, there are two main forms of interactions that can be standardised. The first one is based on interface specifications while the second revolves around communication protocols. Figure 2.7 illustrates these approaches.
In (a) the standard is exposed through a set of specified interfaces, typically realised as Application Programming Interfaces (API). The syntax and semantics of each function are defined. Standards based on interface specifications guarantee application portability at the source code level. For example, applications can be ported to all other conforming implementations and expect to exhibit the same behaviour. The POSIX [38] standards fall into this category.

![Diagram of Interface and Protocol](image)

**Figure 2.7 Forms of interaction**

In (b) the standard is visible through the specification of protocols. Communicating parties exchange messages in a defined format and procedure. Since the standardised messages appear at the level of transmission, interoperability between conforming systems is guaranteed. Further, this facilitates maximum flexibility and optimisation in implementing a conforming system. Although the message syntax and semantics are defined, access to the service is usually not. This prevents applications using this kind of standard from being portable across different conforming implementations. The ISO/IEEE medical devices communication [39] standard is an example.

One key decision to make is how the standard is exposed to the parties using it. A decisive usage scenario for carrier-grade platforms in network element application development is the porting of applications across a variety of deployment hardware. Standardising through interface specification for the service availability middleware is therefore appropriate due to the guarantee of application portability. In addition, it makes application integration with other subsystems easier when the functions are defined.
2.6 Summary

This chapter introduces the constituent components of a typical mobile network. It explains the hierarchical relationship between the end users, mobile network operators and network equipment providers. The challenges faced by the network equipment providers include a long list of requirements passed downwards by the end users and mobile network operators, and the expectations to have an ever shorter time-to-market at lower development costs.

These challenges can potentially be resolved by following the principle of a software product line approach. The application support of service availability in a carrier-grade platform is considered to be one of the key reusable assets across different members of a product family. It consists of availability management and online software upgrade, which are the two necessary application support for minimising service outage. Configurable redundancy models and upgrade methods have been identified as the variation points for these reusable assets.

Implementing these identified reusable assets using standardised components has potentially many benefits. Not only does it decreases the risks of getting locked-in by a single vendor, but also reduces the overall complexity of the final system. Having a standards-based carrier-grade platform enables a network equipment provider to move up the value chain by branching out to new business opportunities.

There are two main forms of interactions that can be standardised. The first one is based on interface specifications and its standardisation guarantees application portability at the source level. The second approach revolves around the protocol between communicating parties and its standardisation guarantees application interoperability. A decisive usage scenario for carrier-grade platforms in network element application development is the porting of applications across a variety of deployment hardware. Therefore, having the service availability standard in the form of interface specification is more applicable. This approach also makes application integration with other subsystems easier.
3 Standards-Based Service Availability Middleware

This chapter argues for the adoption of standards-based availability management middleware as a building block in carrier-grade platforms and discusses what constructs are worth standardising. It gives an overview of the Service Availability Forum’s Application Interface Specification and Hardware Platform Interface standards, and explains the process used for developing its specifications. This chapter is based on the included publication [P3].

3.1 Why a Service Availability Standard?

The main benefits of adopting a standards-based availability management middleware in a carrier-grade platform are:

- **Ease of integration.** Due to the ongoing convergence of communications and information technology, integration of various components is made simpler when there is a standard clearly defining the boundary of service availability. This enables a product to evolve and adapt to other convergent technologies in the future.

- **Risk management.** Adopting a standards-based approach encourages the development of COTS components in the industry. This avoids the risk of vendor lock-in because there are typically more than one supplier. In addition, a standard offers a consistent basis for comparing service availability features in different systems, and ensures that compatibility and interoperability among different products from different suppliers.

- **Costs reduction.** Developers of a network element product can focus on the core application developments. The resulting applications are portable across different hardware and operating systems by reusing the standardised service availability capabilities in all products. Costs of maintenance are reduced when compared with proprietary service availability solutions, which are very often ad hoc, narrowly focused and vertically integrated with the entire network infrastructure device on a per-product basis.

- **Organisation.** From the organisation’s perspective, a standard improves the supply of staff with relevant service availability experience. This also reduces expenses associated with training and education of technical and operational personnel.

All of these benefits, when realised in the network element product development, contribute considerably to the shorter time-to-market at lower development costs challenge.
faced by the network equipment providers, although they are by no means the sole solution to meeting the expectations and requirements as shown in Figure 2.2 on page 13.

3.2 Building Blocks

One of the most difficult tasks in creating a standard is to determine where its boundary is, with reference to the rest of the system. Standardising at the appropriate level of abstraction is important and has a significant impact on its acceptance. If a standard is at a very high level of abstraction, its usefulness will be limited. For example, specifying that a standard must have a database would enable plenty of systems to claim compliance to such a standard but in reality, there is no guarantee of portability or interoperability among these systems. Lowering the level of abstraction to the other extreme however will result in prescribing how a system ought to be designed. A delicate balance must also be struck to leave room for competition under product differentiations such as quality of implementation, performance, and costs.

Standardising service availability for carrier-grade platforms is complex. Not only are there many variations of the commonly used service availability mechanisms, but also the selection of which one to use always lies in the application where the product requirements are known. One approach is to standardise the key service availability mechanisms as building blocks. A product developer selects the appropriate building blocks based on the specific requirements and creates the network element application accordingly.

The key service availability building blocks for network element applications have been identified as:

- Means of maintaining information about software entities and their relationships. In order to manage the availability of software entities, there needs to be a representation showing the kinds of software that exist in the system, how they are related to each other, their availability status, and which ones are involved in what kind of redundancy. Version information related to each software entity is required for online software upgrade.

- Means of configuring protective redundancy. As a solution residing on a platform, a systematic way of managing and implementing variations of a range of protective measures such as redundancy model is needed.

- Means of managing faults from their detection to recovery and repair. The service availability building block is expected to provide the mechanisms of detecting and monitoring errors of a software entity; automatic recovery operations such as restart and failover actions; and repair actions such as reloading a software entity and rebooting a node.

- Means of replacing and upgrading software entities. There must be a way in which an upgrade can be described based on the current deployment configuration and the
desired one. In order to provide online software upgrade with minimum or no service interruption, failure detection and handling during the upgrade operation must be present.

- Means of configuring upgrade strategies. As a solution residing on a platform, a systematic way of managing and implementing variations of different upgrade methods, spheres of upgrade and service outage duration is needed.

3.3 Service Availability Forum Middleware

When the Object Management Group (OMG) was developing the Fault Tolerant CORBA (Common Object Request Broker Architecture) middleware standard [61], the network equipment providers widely perceived it as an enabler for reducing development costs and effort by simply buying service availability capabilities from third-party suppliers instead of building them in-house [P2]. However, the resulting specification tries to be generic across wide-ranging applications from the enterprise domain to the communications and mission critical systems, which have very different dependability requirements. As a consequence, the interface definitions do not contain the required support for the specific needs of telecommunications applications, including the essential interaction with other well established standards, notably state management [15], alarm reporting [16] and event management [17]. As a result, there is a need to develop an open standard for service availability specifically for telecommunications [P3] in the Service Availability Forum (SAF) [74], which is an industrial coalition with a mission to foster an ecosystem that enables the use of COTS building blocks in the creation of highly available network infrastructure products, systems and services.

![Service Availability Forum middleware](image)

**Figure 3.1 Service Availability Forum middleware**
Figure 3.1 illustrates the relative position of the Service Availability Forum middleware in a network element as shown in Figure 2.3 on page 15. The middleware implements the corresponding high availability function and it is made up of the **Application Interface Specification** (AIS) [80], and **Hardware Platform Interface** (HPI) [79]. AIS is closer to the network element application and aims to provide support for developing highly available applications in a distributed computing environment, while HPI is nearer to the underlying hardware platform and primarily deals with the management of the physical components of a carrier-grade computing platform.


### 3.4 Application Interface Specification

AIS abstracts the high availability characteristics into a conceptual model upon which an implementation provides standard methods to the application developers to respond and manage events such as failures. AIS defines an extensive set of APIs for both threaded and non-threaded applications, supporting synchronous and asynchronous interfaces.

Central to the support for developing highly available applications is the Availability Management Framework, which is a software entity that ensures service availability by coordinating other software entities within a computer cluster. It provides a set of functions, which includes error reporting and health monitoring, to enable highly available applications development. A more thorough explanation of this Framework will be given in Chapter 4.

Working with the Availability Management Framework, the Software Management Framework carries out online software upgrades by orchestrating the migration from one deployment configuration to a desired one with no or only minimum service interruption. It provides a means to describe how the changes should take place, together with a set of functions for error monitoring and recovery. Chapter 5 discusses this Framework in more details.

Additional services, known as AIS Services, have also been defined in AIS to support the development of a carrier-grade system. However, an application does not necessarily need to use all these services. For completeness, a brief description of each service is included in Table 3.1.
Table 3.1 AIS services

<table>
<thead>
<tr>
<th>Service</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Checkpoint Service</td>
<td>Facilitates the recording of checkpoint data incrementally to protect an application against failures.</td>
</tr>
<tr>
<td>Cluster Membership Service</td>
<td>Provides applications with the current membership information about the nodes in a cluster.</td>
</tr>
<tr>
<td>Event Service</td>
<td>Offers an asynchronous communication means between multiple publishers and multiple subscribers over an event channel.</td>
</tr>
<tr>
<td>Information Model Management Service</td>
<td>Manages all the objects in the Information Model, which is essentially a representation of the current system configuration of the entities constituting a Service Availability Forum cluster, and provides an interface to access and manage these objects.</td>
</tr>
<tr>
<td>Lock Service</td>
<td>Synchronises access to shared resources among application processes on different nodes in a cluster.</td>
</tr>
<tr>
<td>Log Service</td>
<td>Records cluster-significant, function-based information for system administrators to resolve issues such as mis-configurations and network disconnects.</td>
</tr>
<tr>
<td>Message Service</td>
<td>Delivers a buffered message passing system based on the concept of a message queue, which preserves messages during a switchover. Message queue groups can be formed by putting message queues together as a mechanism for masking the failure of a receiver process.</td>
</tr>
<tr>
<td>Naming Service</td>
<td>Provides a mechanism for binding human-friendly names to service access points, communication end points and other resources that provide some sort of service. This allows for a service provider to advertise a service and a service user to perform a lookup.</td>
</tr>
<tr>
<td>Notification Service</td>
<td>Primarily used by a service user to report an event or a change of status to a peer service user in fault management.</td>
</tr>
<tr>
<td>Timer Service</td>
<td>Signals to an application when either an absolute time or a duration expires.</td>
</tr>
</tbody>
</table>

3.5 Hardware Platform Interface

By abstracting the platform specific characteristics into a conceptual model, an HPI implementation provides the users with standard methods of monitoring and controlling the physical hardware. A platform management application that uses HPI is therefore portable across systems. The abstraction is based on the notion that a user opens a session on a domain of resources, discovers the management capabilities associated with the entity that is to be managed, and then interacts with the entity through standardised functions. Table 3.2 explains the basic HPI concepts.
Table 3.2  HPI basic concepts

<table>
<thead>
<tr>
<th>Concept</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entity</td>
<td>Represents a physical component to be managed in a system.</td>
</tr>
<tr>
<td>Resource</td>
<td>Logical representation of one or more entities that share a common management access through the management instruments and management capabilities mechanisms.</td>
</tr>
<tr>
<td>Domain</td>
<td>Collects a group of related resources, each of which provides access to information about the entity it represents.</td>
</tr>
<tr>
<td>Session</td>
<td>Provides user access to one domain for management at a time. There may be multiple sessions opened for an user at any one time.</td>
</tr>
</tbody>
</table>

A management instrument models the manageability of each entity in a resource, and provides a set of uniform mechanism for receiving and controlling the state of an individual entity. Table 3.3 explains the different types of management instruments.

Table 3.3 Management instruments

<table>
<thead>
<tr>
<th>Instrument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sensor</td>
<td>Provides a reading of the operational or health data of an entity.</td>
</tr>
<tr>
<td>Control</td>
<td>Reflects the current state set for an entity and allows for its mode to be changed.</td>
</tr>
<tr>
<td>Inventory Data Repository (IDR)</td>
<td>Stores descriptive information about each entity, which may include serial numbers, part numbers, manufacturing dates etc.</td>
</tr>
<tr>
<td>Watchdog Timer</td>
<td>Monitors an entity’s health by providing configurable actions to be performed when a watchdog timer expires.</td>
</tr>
<tr>
<td>Annunciator</td>
<td>Communicates an entity’s fault conditions and other status information through a set of announcements.</td>
</tr>
<tr>
<td>Diagnostic Initiator Management Instrument (DIMI)</td>
<td>Offers an uniform interface for invoking diagnostic tests of individual entities in a resource.</td>
</tr>
<tr>
<td>Firmware Upgrade Management Instrument (FUMI)</td>
<td>Presents an abstracted, entity-independent firmware upgrade sequence for each entity in a resource.</td>
</tr>
</tbody>
</table>

A management capability is associated with the manageability of a resource as a whole. It provides a means to getting and setting the desired capability of a resource. Table 3.4 describes the various management capabilities of a resource.
Table 3.4  Management capabilities

<table>
<thead>
<tr>
<th>Capability</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Managed Hot Swap</td>
<td>Indicates that a resource is able to generate the required events when it is inserted or extracted from the system during normal operation.</td>
</tr>
<tr>
<td>Configuration</td>
<td>Provides the support for transitioning between the current operational, saved and factory default settings.</td>
</tr>
<tr>
<td>Load Management</td>
<td>Identifies and allows for changing the software to be loaded onto the entity associated with the resource.</td>
</tr>
<tr>
<td>Reset Management</td>
<td>Supports for selecting either a warm (preserving states) or cold reset.</td>
</tr>
<tr>
<td>Power Management</td>
<td>Allows for setting the state to be powered on, powered off or power-cycled.</td>
</tr>
</tbody>
</table>

Figure 3.2 illustrates an example HPI conceptual model. A management application interacts with an HPI implementation using the standardised functions of session management and domain discovery for establishing sessions and discovering the domains of the system to be managed. Events and alarms generated by the entities and resources in the domain associated with a session are propagated back to the management application via the corresponding functions.

Figure 3.2  HPI conceptual model
In this example, there are M domains, each of which has a number of resources. Each resource provides the management application with access to the appropriate management instruments that manage the various physical components represented as entities in the domain. For each resource in a domain, there is one set of management capabilities that are applicable to the resource as a whole.

3.6 Creating a Service Availability Forum Specification

This section gives an overview of how the Service Availability Forum specifications are generally created by discussing the specification development process. The description follows the spirit of the process instead of providing the exact details, however.

The development of a Service Availability Forum specification follows a four-step process as shown in Figure 3.3. Each step generates a particular deliverable after an affirmative decision. They are:

1. Request For Proposal creation. A Request For Proposal (RFP) is a document that contains the scope, assumptions and requirements of a particular area of development. Individual member companies work together on the RFP, which is then reviewed by the architecture group for consistency with other published specifications, technical accuracy and management of trade-offs in scope, quality and time to publication. The process of review and rework iterates until consensus is reached to issue the RFP. No formal voting is conducted in this step.

![Figure 3.3 Specification development process](image-url)
2. Proposal generation. A specification development group is formed by member companies who are interested in responding to the RFP. How the group works together is an internal matter but typically involves weekly conference calls, regular face-to-face meetings and status updates to the general membership. The proposal is reviewed by an evaluation team nominated by the Forum. Again, the review and rework process iterates until consensus on all technical issues is reached. No formal voting is conducted in this step. The approved proposal can be considered as technically sufficient for addressing the RFP requirements.

3. Draft specification production. An appointed editor takes over the drafting of the specification based on the approved proposal, following a consistent style and language used in released specifications. Working closely with the editor, the specification development group iterates the review and rework process until the criteria for releasing the draft specification are met.

4. Specification finalisation. At the specification finalisation stage, formal voting is conducted by the general membership and the Board of Directors. The voting for the two occasions are the same in that there is one vote per member company. Quorum is at least two thirds of the eligible voting members casting valid votes. Approval are by super-majority, that is, approved by at least two thirds of the votes cast. If any of this vote fails, the draft specification is sent back for rework. Otherwise, the draft specification is approved and released to the public as an official specification.

   The specification development process is rigorous involving many reviews and refinements. And yet, the combination of formal voting and reaching consensus by all the stakeholders ensures that the developed specifications are relevant, timely and applicable in the mobile network industry.

   The time required for creating a specification varies. As an indication, it took 21 months (August 2001 – April 2003) for the first release of the Availability Management Framework and three years (October 2004 – October 2007) for the Software Management Framework.

3.7 Summary

The benefits of having a service availability standard are ease of integration, risk management, costs reduction and organisation. The constructs that are worth standardising are the key parts of a highly available system. They are the methods for storing information about components and their relationships; configuring protective redundancy; managing faults from detection through recovery and repair; and replacing and upgrading components.

   The OMG Fault-Tolerant CORBA specification tries to be generic across wide-ranging applications and does not contain the required support for the specific needs of
telecommunications applications, including the essential interaction with other well established standards has led to the formation of the Service Availability Forum to standardise service availability capabilities for the mobile network industry.

The Service Availability Forum middleware implements the high availability function of a network element. The middleware consists of AIS and HPI. The former aims to provide support for developing highly available applications in a distributed computing environment, while the latter deals with the management of the physical components of a carrier-grade computing platform.

The specification development process in the Service Availability Forum is rigorous involving many reviews and refinements. The combination of formal voting and reaching consensus by all the stakeholders ensures that the developed specifications are relevant, timely and applicable in the mobile network industry.
4 Availability Management Framework

This chapter examines the solution for supporting availability management of applications in carrier-grade platforms, which has been contributed by the author and formed part of the Availability Management Framework (AMF) specification standardised by the Service Availability Forum. An example GGSN that uses the AMF solution is shown as a case study. It illustrates the feasibility of reusing the availability management support offered by AMF in a software product line. This chapter is based on the published Availability Management Framework specification [76] and the included publication [P2].

4.1 Overview

The Availability Management Framework (AMF) [76] is a software entity that provides an application with service availability support by coordinating redundant resources within a cluster of interconnected nodes.

AMF defines a basic model that captures the relationships among the entities to be managed. The management of these entities requires that their conditions, known as states be understood. Protective measures are made available according to a redundancy model that expresses how the additional resources can be organised. A component capability model provides an application developer with the ability to defer the decision of which redundancy scheme to use until deployment time. The following sub-sections explain the basic model, component states, redundancy model and component capability model of AMF.

4.1.1 Basic Model

A resource is defined as a physical entity managed by AMF, which includes hardware and software processes. The latter is regarded as being equivalent to the definition presented by the POSIX standards [38]. A physical node is expected to behave like a computer. Nodes are interconnected by some form of communication media. A local resource is contained within a physical node. This implies that all contained resources would become inoperable if the physical node fails, including for example hardware attached to or software processes running on a physical node. An external resource is independent of the operation of a physical node, for example, intelligent I/O board in a blade chassis. Failures of these resources are independent of physical node failures.

In order to manage resources in a system, AMF uses an abstract conceptual model to represent the resources under its control as logical entities. All logical entities are identified with unique names with their attributes, relationships and their mappings to the physical resources.
pre-configured into a system model and stored in a configuration repository. Table 4.1 presents the AMF logical entities.

Table 4.1 AMF logical entities

<table>
<thead>
<tr>
<th>Entity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cluster</td>
<td>A collection of nodes that may join and leave at any time.</td>
</tr>
<tr>
<td>node</td>
<td>A logical representation of a physical node.</td>
</tr>
<tr>
<td>component</td>
<td>A set of resources managed by and viewed from AMF for error detection and isolation, recovery and repair. An example is a piece of software.</td>
</tr>
<tr>
<td>local component</td>
<td>A subset of local resources contained within a single physical cluster node.</td>
</tr>
<tr>
<td>external component</td>
<td>Represents a set of resources that are external to the cluster.</td>
</tr>
<tr>
<td>SA-Aware Component</td>
<td>Under the direct control of AMF and provides an interface for availability and state management (e.g. workload assignment and removal.) It may comprise of one or more processes.</td>
</tr>
<tr>
<td>Non-SA-Aware Component</td>
<td>Do not register directly with AMF. These are typically legacy or existing applications. For example, system resources such as networking and storage implemented by the operating environment do not need any process. Components representing only local hardware resources would have added unnecessary complexity. Another category is complex applications such as databases or application servers, which are likely to have their own availability management.</td>
</tr>
<tr>
<td>Component service instance (CSI)</td>
<td>Represents the workload that AMF can dynamically assign to a component. High Availability (HA) states (in Section 4.1.2) are assigned to a component on behalf of its CSI. It has a set of attributes that are made up of name/value pairs, which are not used by AMF but are passed to the components. CSIs having the same type mean that they share the same list of attribute names. Multi-valued attributes are supported by allowing the same name to appear several times.</td>
</tr>
<tr>
<td>Service Unit</td>
<td>An aggregation of a set of components combining their individual functionalities to provide a higher level of service. It contains one or more components but a component can only be in one service unit at any given time. It is defined at deployment time. Local components and external components cannot be mixed within a service unit.</td>
</tr>
<tr>
<td>Service Instance</td>
<td>An aggregation of CSIs to be assigned to the individual components of the service unit. It represents a single workload assigned to a service unit. When a service unit is available to provide service, AMF assigns zero or more service instances by assigning each individual CSI of the service instance to a specific component within the service unit. When a service unit becomes unavailable, AMF removes all the previously assigned service instances. When the assignment of CSIs to components is considered, a component must support the same CSI type. The actual assignment within a service unit is not dictated by AMF. A component may have zero or more assignments.</td>
</tr>
<tr>
<td>Service Group</td>
<td>Contains one or more service units, each of which contains the components of specific component types, and provides service availability for one or more service instances. From the AMF perspective, service unit is the unit of redundancy. It has a redundancy model defining how the service units in the service group are associated to provide service availability.</td>
</tr>
</tbody>
</table>
Figure 4.1 illustrates a cluster with three interconnected nodes, X, Y and Z. Software processes that are contained in each node are local resources. For example, node X contains local resources that include software processes for components A1, B1 and C1. The service provided by this cluster is configured as two service groups, SG1 and SG2.

SG1 contains two service units, SU1 and SU2 of type SUTypeA. SU1 consists of two components, A1 and B1 of types A and B respectively, which typically represent software processes that are running on node X. Similarly, SU2 contains two different instances, A2 and B2 of types A and B, and run on node Z.

SG2 is made up of three service units of type SUTypeB, each of which has one component of type C running on nodes X, Y and Z.

The workloads of this cluster are represented by service instances SI-1, SI-2 and SI-3. SI-1 is an aggregate of the component service instances A-CSI and B-CSI, which can be served by components of types A and B. SI-2 and SI-3 has only one CSI each, C1-CSI and C2-CSI respectively. In this example, components of type C can service both types of CSI.

For each service instance, there is typically one service unit that handles the live workload and another service unit that acts as a standby. Since the SIs are decomposed to their constituent CSIs, the corresponding status is carried to the CSI level. The active and standby status are indicated as solid and dashed arrows in the diagram.
Service groups and service units are shown enclosed in dotted lines because they are administrative constructs, whereas nodes and components are bordered by solid lines as they represent physical resources.

4.1.2 Component States

AMF defines different states associated with all the logical entities including nodes, service groups, service units, components, service instances and component service instances. However AMF’s APIs only provide management of a subset of the defined states for components and CSIs. The remaining states are relevant to System Management functions. Therefore, the discussion will only concentrate on the state models for components.

The overall state of a component is a combination of its underlying states: presence, operational, readiness and HA state per CSI. Table 4.2 describes each of the states.

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>presence</td>
<td>This state reflects a component’s life cycle and reveals whether it is in the state of uninstantiated or instantiated, or in the process of instantiating, restarting or terminating. It also covers the situations when an entity cannot be instantiated after all possible retries or simply fails to terminate.</td>
</tr>
<tr>
<td>operational</td>
<td>This state refers to the ITU X.731 state management model [15] and shows that whether the component is operable or not, indicating to AMF if it is capable of taking CSI assignments.</td>
</tr>
<tr>
<td>readiness</td>
<td>This state indicates whether a component is available to take up CSI assignments and is used by AMF for making assignment decisions. A component can be in out-of-service, in-service or stopping state.</td>
</tr>
<tr>
<td>HA state</td>
<td>Central to the availability management support is the HA state of a component per component service instance. For a component supporting the full HA state, it has to implement the full transitions as shown in Figure 4.2.</td>
</tr>
</tbody>
</table>

When AMF assigns the active HA state to a component on behalf of a CSI, the component must start to provide service characterised by that CSI. When the HA state is standby, the component must prepare itself for a swift and smooth takeover the active role should AMF requests so. When the quiesced HA state is given, the component must get the work into a condition such that it can be transferred to another component with as little service disruption and as quickly as possible. In this way, AMF can safely assign the active HA state for this CSI to another component. As soon as the quiescing HA state is assigned, the component must reject attempts from new users to access the service and only continue to serve existing users until they all finish using it. At this point the component notifies AMF of this fact.
The last two HA states are specific to telecommunications applications. The transition from active to quiesced state is typically used in switchover scenarios in which a healthy active component is replaced by another component as a result of administrative operations or escalation of recovery procedures. The goal is to minimise the potential service disruption in this situation. The transition from active to quiescing state, then implicitly onwards to quiesced state is usually the consequence of a shutdown administrative operation.

The events add, remove and change as shown in the diagram correspond to the adding, removing and changing the status of a CSI assignment to a component, and represents all the possible transitions from one state to another.

**4.1.3 Service Group Redundancy Model**

The service group redundancy models are defined in terms of the rules followed by AMF when assigning the active and standby HA state to service units of a service group for a service instance. The defined redundancy models are 2N, N+M, N-way, N-way active and no redundancy.

Figure 4.3 on the next page shows that service group SG1 has a 2N redundancy model, which is effectively a 1+1 active/standby configuration that can be repeated N times. Under normal operation in (a), service instance SI-1 is an active assignment to service unit SU1, while service unit SU2 has the standby assignment of SI-1. The corresponding active workload

---

**Figure 4.2 State diagram of the HA state for a CSI**

Add – add a component service instance assignment
Remove – remove component service instance assignment
Change – change the HA state of a component service instance assignment

---

[Diagram of HA state transitions]
assignments are propagated to components A1 and B1 on node X. At the same time, components A2 and B2 are assigned the standby CSIs on node Y.

When node X fails as illustrated in (b), AMF fails over all the previously active workload assignments in service unit SU1 to SU2. No standby assignments are performed as there are no more redundant node configured into this cluster. At this point, SG1 is unprotected from further failures.

Figure 4.3 2N redundancy model

Figure 4.4 shows a N+M redundancy model, where N and M are the number of service units that active and standby HA states of service instances can be assigned respectively. In this example, all the active assignments are directed to the service units SU1, SU2 and SU3 (N=3) configured on nodes V, W and X. The standby service instances are assigned to service units SU4 and SU5 (M=2) on nodes Y and Z. If SU1 on node X fails, the service instance SI-1 will be assigned the active state to the previously standby SU4 on node Y. Since in a N+M redundancy model active and standby assignments cannot be mixed, the standby assignment of SI-3 to SU4 must now be removed. How the new standby assignment of SI-3 is performed is implementation dependent and outside the scope of the specification. The aim of this redundancy model is to maintain as many active service instance assignments as possible.
In a N-way redundancy model, N service units can be simultaneously assigned a mixture of active and standby service instances. This is similar to the N+M redundancy model except that no service units are designated to hold only active or standby assignments. By allowing for mixed active and standby assignments to a service unit, a N-way redundancy model can potentially achieve what a N+M redundancy model does with fewer service units. This is at the cost of more complex implementation of a component that must support both the active and standby assignments at the same time. If a service unit on a node fails, the active assignments will be assigned to the service units that are currently designated as their standbys. The standby assignments will also be moved to other service units. In case there are not enough resources on the remaining service units for all these re-assignments, some of the standby assignments will be removed and this effectively puts those affected active assignments into a non-redundant mode. If further resources are reduced due to additional failures, some of the lower ranked active assignments have to be dropped from the system.

In a N-way active redundancy model, N service units can only be assigned active service instances. For each service instance, it can be assigned to multiple service units. The protection is achieved by servicing a workload in all the assigned active service units simultaneously. The assumption is that in a failure situation, there may be one service unit that continues to provide the service without delay. This redundancy model is basically an active replication of services and is typically used when the service unavailability time is less than the time required for failure recovery in 2N, N+M and N-way redundancy models. The use of this redundancy model is expensive and due care must be taken to ensure that active components are in agreement in spite of nondeterminism.

The no-redundancy model means that a service unit is assigned the active HA state for at most one service instance but never a standby. This implies that no two service units exist having the same service instance assigned to it. If a service unit fails, the assigned service
instance will not be served until there is a new service unit introduced into the service group. Restarting a service unit is the only recovery action under this redundancy model and it is typically used with non-critical components.

4.1.4 Component Capability Model

By focusing on the workload in the AMF basic model, a developer does not need to include the explicit logic of the chosen redundancy model into their design during build time. The principle behind this is to defer the redundancy model decision until deployment time. A *component capability model* has been defined and the classifications are shown in Table 4.3.

<table>
<thead>
<tr>
<th>Model</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>x_active_and_y_standby</td>
<td>A component supports all values of the HA states and can be assigned multiple active and multiple standby CSIs.</td>
</tr>
<tr>
<td>x_active_or_y_standby</td>
<td>A component supports all values of the HA states and can be assigned either multiple active or multiple standby CSIs.</td>
</tr>
<tr>
<td>1_active_or_y_standby</td>
<td>A component supports all values of the HA states and can be assigned either one active or multiple standby CSIs.</td>
</tr>
<tr>
<td>1_active_or_1_standby</td>
<td>A component supports all values of the HA states and can be assigned either one active or one standby CSI.</td>
</tr>
<tr>
<td>x_active</td>
<td>A component can be assigned multiple active but no standby CSIs.</td>
</tr>
<tr>
<td>1_active</td>
<td>A component can be assigned one active but no standby CSI.</td>
</tr>
</tbody>
</table>

Table 4.4 shows the component capability model and service group redundancy model mapping. In principle, the shaded entries could have been marked with Xs according to the service group redundancy model definition. However, without the capability to act as a standby, a component cannot truly take part in the 2N nor N+M redundancy models.
Table 4.4 Component capability model and service group redundancy model mapping

<table>
<thead>
<tr>
<th>Component capability model</th>
<th>2N</th>
<th>N+M</th>
<th>N-way</th>
<th>N-way active</th>
<th>no-redundancy</th>
</tr>
</thead>
<tbody>
<tr>
<td>x_active_and_y_standby</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>x_active_or_y_standby</td>
<td>X</td>
<td>X</td>
<td>-</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>1_active_or_y_standby</td>
<td>X</td>
<td>X</td>
<td>-</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>1_active_or_1_standby</td>
<td>X</td>
<td>X</td>
<td>-</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>x_active</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>1_active</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

4.1.5 Programming and Administrative Interfaces

AMF provides two types of interfaces, one at the user program level and the other is for the system administrators.

The programming interfaces allow a user program to interact with AMF. Table 4.5 contains a brief description of the AMF API grouped under nine categories.

Table 4.5 Availability Management Framework API

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Library life cycle</td>
<td>Function calls for connecting and disconnecting with the AMF library.</td>
</tr>
<tr>
<td>Component registration and unregistration</td>
<td>Function calls for registering and unregistering components with AMF.</td>
</tr>
<tr>
<td>Passive monitoring</td>
<td>Function calls for requesting the AMF to perform passive monitoring on a component’s processes.</td>
</tr>
<tr>
<td>Component health monitoring</td>
<td>Function calls for requesting AMF to perform healthchecks on components.</td>
</tr>
<tr>
<td>Component service instance management</td>
<td>Function calls for adding, removing and quiescing CSIs.</td>
</tr>
<tr>
<td>Component life cycle</td>
<td>Function calls to request a component to terminate.</td>
</tr>
<tr>
<td>Protection group management</td>
<td>Function calls to request AMF to start and stop the tracking changes in a protection group.</td>
</tr>
<tr>
<td>Error reporting</td>
<td>Function calls to report and clear an error of a component to AMF.</td>
</tr>
<tr>
<td>Component response</td>
<td>A generic response call to AMF reporting result of a previous AMF request.</td>
</tr>
</tbody>
</table>
A set of AMF Administrative API has been defined for the system administrators, which is shown in Table 4.6.

### Table 4.6 Availability Management Framework Administrative API

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unlock</td>
<td>An operation to lock a logical entity.</td>
</tr>
<tr>
<td>Lock</td>
<td>An operation to unlock a logical entity.</td>
</tr>
<tr>
<td>Lock instantiation</td>
<td>An operation to cause a logical entity to become non-instantiable after its termination.</td>
</tr>
<tr>
<td>Unlock instantiation</td>
<td>An operation to cause a logical entity to become instantiable.</td>
</tr>
<tr>
<td>Shutdown</td>
<td>An operation to shutdown a logical entity.</td>
</tr>
<tr>
<td>Restart</td>
<td>An operation to restart a logical entity.</td>
</tr>
<tr>
<td>SI swap</td>
<td>An operation to interchange the states of two service instances.</td>
</tr>
<tr>
<td>SG adjust</td>
<td>An operation to restore the initially stated preferred service instance assignments in a service group.</td>
</tr>
<tr>
<td>Repaired</td>
<td>An operation to clear the disabled operation state of a logical entity after it has been repaired.</td>
</tr>
<tr>
<td>EAM start</td>
<td>An operation to start external active monitoring (EAM) on a logical entity.</td>
</tr>
<tr>
<td>EAM stop</td>
<td>An operation to stop external active monitoring (EAM) on a logical entity.</td>
</tr>
</tbody>
</table>

### 4.2 Architecting an Example GGSN

This section explains what a Gateway GPRS Support Node (GGSN) is and then presents a design that uses AMF to support the availability management in the application.

GGSN is a key network element that is responsible for interconnecting a mobile user’s device with a PDN such as the Internet. This is a representative case as the network element bears all the characteristics of communication and information technology convergence, an area where mobile network equipment providers strive to innovate as a means to be ahead of the competitors.

GPRS resides in the Packet Switched domain of a mobile network and Figure 4.5 shows the logical architecture of GPRS. This is a much simplified architecture showing only the context of the discussion. Readers are referred to the GPRS specification [1] for a full description of all the systems interacting with the Packet Switched domain core network, together with the standard interfaces among them.
GPRS contains functionality required to support several levels of quality of service of data transfer between a Mobile Station (MS) and a Packet Data Network (PDN). A PDN may be either an IP-based intranet or the Internet. An MS accesses the GPRS service via a Base Station System (BSS), which in turn connects to the Packet Switched domain core network via a Serving GPRS Support Node (SGSN). The SGSN keeps track of the location of an individual MS and performs security functions and access control. The Gateway GPRS Support Node (GGSN) provides internetworking with a PDN and is the first point of interconnection with a Public Land Mobile Network supporting GPRS. A GGSN is connected to a SGSN by an IP-based packet domain backbone network. The Home Location Register (HLR) contains GPRS subscription data and routing information, which is accessible from both the GGSN and SGSN. The Charging Gateway Functionality (CGF) collects charging records according to the amount of data transferred, the quality of service supported and the duration of the connection from the SGSN and GGSN.

In order to send and receive packet data, an MS must activate a Packet Data Protocol (PDP) context that it wants to use. A PDP context contains information such as the subscriber’s identity, type of packet data network, packet data network address, quality of service profile, charging characteristics, IP address of the SGSN currently serving this MS, to name just a few. This makes the MS known in the corresponding GGSN before internetworking with a PDN can commence. User data are then transferred transparently between an MS and a PDN using encapsulation and tunnelling, eliminating the need for a packet domain core network to interpret external data protocols.
Based on the defined mapping of GPRS functions to the logical architecture, a GGSN typically performs message screening and charging data collection under Network Access Control; relay, routing, address translation and mapping, encapsulation and tunnelling under Packet Routing and Transfer; and Mobility Management. However, in the example only the functions of PDP context management (which includes activation, modification and deactivation), PDP address allocation, and charging are shown.

![Figure 4.6 High level design of example GGSN](image)

Figure 4.6 shows the high level design of the example GGSN. In this design each function, namely, PDP context management, PDP address allocation and charging data collection are represented as individual blocks, each of which implements the corresponding defined function according to the specification. The shaded PDP context table is considered to be a piece of information accessible by the three functions. Altogether they form the example GGSN network element application.

By focusing only on the service availability aspect, this simplification reduces the complexity of the example, and yet it still retains the realistic interactions of GGSN to other systems including SGSN, PDN and CGF. Moreover, the purpose of this GGSN example is to illustrate the use of and evaluate the availability management of a network element. Therefore, the discussion does not need to go into the next level of details of the GGSN functional requirements.

Figure 4.7 shows one design of the example GGSN. The GGSN service is modelled as an AMF application and uses 2N redundancy model. Each of the example GGSN function, namely, session management, PDP address allocation and charging data collection is mapped onto AMF components of types PDPMgt, AddrAlloc and Charging respectively. One instance of each of these components is instantiated at deployment time, forming a service unit known as G1 that is hosted on node X. Similarly, a second service unit G2 is created by instantiating the three types of components on node Y.
The workload of the components are represented by their individual component service instances, which are shown as PDPMgtCSI, AddrAllocCSI and ChargingCSI respectively. All these CSIs are aggregated to form the service instance for the example GGSN application providing a service represented by the service group GGSNService.

Figure 4.7 Example GGSN in 2N

The shaded service unit indicates that G1 is an active unit whereas G2 is a standby. Should there be a fault on node X, the GGSN service instance will be failed over by AMF to node Y where the standby G2 will take over and provide the service instead. This is accomplished by transitioning all the previously HA states of PDPMgtCSI, AddrAllocCSI and ChargingCSI from standby to active. Under this situation, the example GGSN example application is left unprotected from this point onwards until the faults have been repaired, or new resources be introduced into the system as protective redundancy.

4.3 Additional Validation

As pointed out in the Research Method discussion in Section 1.4 on page 7, technical evaluation of AMF has already been carried out as part of the specification creation process. This section provides additional validation by referring to a prototype implementation, and an illustration of the level of support offered by AMF to help construct a software product line for the example GGSN presented in Section 4.2 above in practice, thus, strengthening the claim that the AMF solution does tackle practically relevant problems. In addition, the currently known limitations are also discussed.
A prototype based on the above example GGSN design has been implemented. As reported in [41], the finding confirms that using AMF as a standards-based availability management middleware in part of the carrier-grade platform for network element applications is feasible. Indeed, all the typical non-functional requirements expected in a GGSN product have been successfully met. These included:

- There was no single point of failure of constituent components.
- The failure switchover was transparent to remote nodes upon detection of a failed active unit, and there was no loss of data in existing sessions.
- In the presence of failures, application data such as billing were preserved.
- Upon failure detection, the failover time from the active to standby unit was measured to be 30 milliseconds.
- The performance overhead on the active unit when it was synchronising with the standby was between 20-30%.

Since a product line is of strategic importance to a business, an assessment of the support provided by AMF to realise a product line of GGSN network elements is therefore included. Derived from the benefits of a software product line [85][89], the following measures have been formulated to further validate how well the AMF solution, when utilised in a carrier-grade platform, supports a large number of GGSN products in a product line. They are:

- **Variability**
  1. **Configurable redundancy model.** The notion of service instance and its constituent component service instance is one of the keys for the flexibility exhibited in the AMF solution. This is a departure from the traditional approach of assigning a role such as active or standby to a component, which requires a developer to build the redundancy model logic into the application during development time already. For example, if there is a need to change the redundancy model of the previous example GGSN (Section 4.2 on page 44) from 2N to N+1 (say 2+1), the AMF solution would simply require a new service unit of type *GGSN* be created and deployed on a new node by instantiating components of types *PDPMgt, AddrAlloc* and *Charging*. The new example application thus provides the service configured by the service group *GGSNService* with the N+1 redundancy model, and service instances served by two active service units and protected by one standby. Since the AMF solution forces a developer to separate the redundancy model logic from that of the application, and allows for deferring the selection of an appropriate protection scheme until deployment time, there is no change needed in the application code. Without such an AMF support however, the components have to be re-designed and implemented accordingly.
Therefore, a redundancy model is a variation point for an AMF-based carrier-grade platform and the binding time is at deployment.

2. **Adjustable component granularity.** While the design in the previous example GGSN (Section 4.2 on page 44) appears to be intuitive, the AMF solution does provide a spectrum of choices that are appropriate to a variety of deployment strategies, potentially scaling the product line to include a considerable number of products. For example, all the three functions could be deployed as single components and each encapsulated into a single service unit, thus a different redundancy model could be applied to the different parts of the GGSN application. For example, the PDP context management may take N+M, while PDP address allocation might opt for N-Way and charging data collection goes for 2N redundancy models to protect against different failure scenarios based on their application needs. The other end of the extreme is to combine the functionalities of SGSN and GGSN into a single service unit, which is also a valid option according to the GPRS specification. Indeed, there are many more design choices in between, which highlight that the AMF solution provides a means to implement the granularity variations. The traditional, non-AMF approach tends to treat each product in the family as an individual design, thus, restricting the ability to use the same structural design for the other products in the same product line. Therefore, the granularity of an AMF component is a variation point for the carrier-grade platform and the binding time is during development.

- **Reuse**

  1. **Code.** There are two types of code reuse in the example GGSN shown in the previous section. The first is the application code of the GGSN. Since the AMF has been standardised, a new GGSN product for the same product line can be created simply by porting the application code to different deployment hardware. The second type of reuse is the code developed for implementing the transitions of the HA state for a CSI as shown in Figure 4.2 on page 39. Since this is a generic mechanism for implementing both the active and standby roles of components such as PDPMgt, AddrAlloc and Charging, they can be replicated and reused by different products in the same product line. Although in the traditional, non-AMF approach the application code of the GGSN can still be reused, the availability management code would require changes according to the structuring of the constituent components and their corresponding assigned roles in a new product.

  2. **Service availability concept.** By defining the service availability notion in an application independent manner in terms of service instances in the AMF
conceptual model, the support for developing an application can be consistently used in other product lines other than the GGSN example. For example, the basis used for analysing the impact on resource requirements based on different service instance assignments, service group redundancy models, component granularity etc. to support the development of the GGSN example is also applicable to other products such as Push-to-talk over Cellular and Lawful Interception Gateway. It is also argued that AMF ensures the service availability property in an application as deterministic as possible. A number of configuration parameters for the service group redundancy model have been defined to support this notion. For example, the ranked list of service instances by importance is used by AMF to select the least important one to drop if needed. The ranking of service units within a service group is used by AMF to determine the order in which service units are selected for service instance assignments. The auto-adjust option makes sure that the highest ranked available service units are assigned the service instance after a new configuration has been effected, typically as a result of repair. All of these contribute to providing an approximation of determining the service availability a priori.

All of the benefits of using AMF come with the assumption that an application is aware of the service availability middleware, which is not a problem when new products are developed. Legacy systems however can only take limited advantage. Although there is support in AMF for integrating legacy systems by means of proxy components, the sheer size and complexity of these legacy systems is causing some concern. At the time of writing, there has not been any reported experience on proxy components, and therefore it remains to be seen how applicable this approach is in practice.

The AMF specification has also grown in size (394 pages) and complexity. As a result the SCOPE Alliance has established a carrier-grade middleware profile [72] detailing the relative priorities of the features in AMF. A high priority gap has recently surfaced from the industry stating the lack of appropriate statistics and measurements of AMF. These information are critical in the overall capacity planning, performance tuning, overload control, and field diagnostics when deploying network elements. Another known but less important shortcoming of AMF is its disconnect with the Hardware Platform Interface specification, in terms of the inconsistent use of the same terms, notably resource, entity and component, and the lack of mapping between the two standards on events and alarms.

4.4 Related Work

Much research has been conducted in the areas of software architectures and fault tolerant systems to date. The most comprehensive survey thus far on architecting fault tolerant systems
From 39 papers representing 25 approaches has recently been published in [59]. Based on the software architecture and fault tolerance viewpoints, the classification framework aims at comparing the state-of-the-art proposals of handling fault tolerance at the software architecture level. By illustrating AMF through the classification parameters, it can therefore be included as one of the candidates and compared with these approaches by re-using the same framework in a consistent manner. This has also resulted in eliminating the unnecessary direct and ad hoc comparisons of AMF with many of the approaches.

Viewing through the fault tolerance classification parameters, AMF has been identified in the concluding part of the survey. They are service-oriented and dynamic architecture, software product line architectures, Model Driven Architecture and explicit resilience, self-healing, and self-adapting systems.

As a result of the analysis of these current approaches, five future research agenda topics have been identified in the concluding part of the survey. They are service-oriented and dynamic architecture, software product line architectures, Model Driven Architecture and explicit resilience, self-healing, and self-adapting systems.

Viewing through the fault tolerance classification parameters, AMF has been identified in the concluding part of the survey. They are service-oriented and dynamic architecture, software product line architectures, Model Driven Architecture and explicit resilience, self-healing, and self-adapting systems.
projects based on the specification have clearly shown that it has grown out of the prototype stage into industrial strength applications, substantiating AMF’s position under the innovative technologies and tools topic. Other advances have also been reported in [42] to automatically generate AMF compliant configurations and a model-driven framework to support an integrated development, analysis and deployment of AIS service configurations in [47] respectively.

In the standards architecture arena, the Object Management Group’s Fault-Tolerant CORBA [61] is the only alternative to the Service Availability Forum’s AMF. Fault Tolerant CORBA is based on the notion of object group in which services are replicated according to a selected strategy such as request retry, redirection to an alternative server, passive (primary-backup) and active replication. Interfaces for the replication manager, fault manager (faults detection and notification), logging and recovery management have been defined. However, its support for other types of redundancy can be considered to be inadequate. For example, support for N+M redundancy model is missing. The limited mandatory requirements indicate that a compliant implementation does not sufficiently guarantee the functionality needed, nor the interoperability with other subsystems. Furthermore, it attempts to be generic across wide-ranging applications from the enterprise domain to the communications and mission critical systems, which have very different dependability requirements. As a result, the interface definitions do not contain the required support for the specific needs of telecommunications applications.

A critical examination of Fault-Tolerant CORBA systems in [27] has also concluded that the specification does not lend itself readily to complex, real-world applications. The reported experiences highlight that there are some severe limitations of Fault-Tolerant CORBA systems include: seemingly stateless CORBA objects must maintain other state information associated with the CORBA infrastructure, server-side transparency, replica consistency and interoperability issues. Other problems such as supporting the CORBA Component Model, combining Fault-Tolerant CORBA with real-time, security and transactions are still open.

A generic approach that shares the same goal as AMF to separate application-dependent and redundancy-related concerns is described in [92]. It utilises a multi-layered reference architecture, a configuration method and an architecture pattern. The structuring framework defines components and uses coordinated atomic actions as a system structuring tool. The multi-level reference architecture comprises of the application, system support, low system and operating system levels. A simple configuration method allows for the most appropriate scheme to be selected by application developers. The Generic Software Fault Tolerance pattern enables the use of dynamic redundancy and masking redundancy but imposes an application to be designed according to this blueprint. The AMF approach has a similar structuring framework, multi-level reference architecture and configuration method. One point of departure is that AMF does not impose such a pattern, but at the same time it does not prevent an application from
following such a pattern either. Another key difference is that this generic approach is based on object-orientation whereas AMF uses the C programming language model.

4.5 Summary

The solution for supporting availability management in carrier-grade platforms has been standardised by the Service Availability Forum as the Availability Management Framework, which is a software entity that provides an application with service availability support by coordinating redundant resources within a cluster of interconnected nodes. The AMF solution forces a developer to separate the service availability concerns from those of the application, and allows for deferring the selection of an appropriate protection scheme until deployment time. AMF has been defined to support various redundancy models including 2N, N+M, N-way, and N-way active. The mechanism for separating the redundancy model logic from that of the application demonstrates that code developed for the active and standby units can be replicated and reused elsewhere.

As a further validation of the AMF solution, a case study using an example GGSN service utilising AMF as the service availability middleware deploying a 2N redundancy model has been examined. An early implementation has shown that this example design has successfully satisfied all the typical functional and non functional requirements of such a product. Additional validation of using AMF as part of a platform in a software product line has substantiated that the variability and reuse criteria have been met. It is also argued that AMF ensures the service availability property in an application as deterministic as possible. A number of configuration parameters for the service group redundancy model have been defined to support this notion, contributing to providing an approximation of determining the service availability a priori.

By classifying AMF in the framework amongst 39 papers representing 25 approaches in the most comprehensive survey to date on architecting fault tolerant systems, it enables a consistent comparison with the most relevant research. It is also argued that AMF has already made considerable progress in two out of the five future research agenda topics of software product line architectures, and innovative technologies and tools.
5 Software Management Framework

This chapter examines the solution for supporting online software upgrade of applications in carrier-grade platforms, which has been contributed by the author and formed part of the Software Management Framework (SMF) specification standardised by the Service Availability Forum. A case study of using SMF to upgrade the example GGSN as shown in the previous chapter is presented. It illustrates the feasibility of using the SMF capabilities to upgrade software in the field with no, or minimum service interruption. This chapter is based on the published Software Management Framework specification [78] and included publication [P4].

5.1 Overview

The Service Availability Forum has recently released the first version of the Software Management Framework [78].

Online software upgrade has two distinguishable phases. The first is software delivery in which a new version of the software is brought into a system. Although it is not important how the upgrade is delivered, how the upgrade is packaged for the delivery is one of the key concerns from the standardisation perspective. The second phase is software deployment, at which time software entities are migrated from the current deployment configuration into the new, desired one. It is at this stage where the actions of an upgrade may cause service interruptions.

In order to maintain service availability, an upgrade operation typically does not migrate all entities simultaneously, instead, the necessary actions are structured into smaller pieces and carried out in such a way that some level of service is still maintained. SMF has been developed to support this organisation and provide for a consistent, coordinated execution of all the constituent parts in an upgrade operation.

The notion of an upgrade campaign has been formulated to serve as a structuring mechanism, providing for different granularities that suit individual systems. An XML (eXtensible Markup Language) schema [82] has been defined to express the execution flow of an upgrade campaign, including support for dealing with the level of service availability that can be provided. The failure handling capability of SMF inherently addresses the issue of service interruption during an upgrade campaign. In order to deliver coherent behaviour, SMF needs a well defined state model to capture all the normal and failure conditions of a system undergoing an upgrade, together with the necessary actions that transition through them. A set of interfaces is also required for coordinating with upgrade-aware applications and the system administrator.
The following sub-sections explain the upgrade campaign, upgrade campaign specification, failure handling, state models, and the programming and administrative interfaces in SMF.

5.1.1 Upgrade Campaign

In the software deployment phase, an upgrade campaign describes the steps and procedures of the migration, which typically transitions through multiple stages according to a system specific, pre-defined sequence. In order to maintain service availability, this sequence tends to follow a pattern of deactivating a part of the system, upgrading the now disabled part and then activating the new entities. Table 5.1 describes the concepts associated with an upgrade campaign.

Table 5.1 Concepts associated with an upgrade campaign

<table>
<thead>
<tr>
<th>Concept</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>upgrade campaign</td>
<td>Defines the steps and procedures for migrating a system from its current deployment configuration into a new, desired one. It is handled as a transaction [32], primarily due to the similar ACID (Atomicity, Consistency, Isolation, Durability) properties required in an upgrade operation. This simplifies the solution for failure handling during an upgrade campaign, which will be shown in section 5.1.3.</td>
</tr>
<tr>
<td>upgrade procedure</td>
<td>Iterates the same collection of actions over a set of software entities. An upgrade campaign may have multiple upgrade procedures, each of which is assigned an execution level. Procedures of the lower levels are executed before the higher level ones, but the ones of the same level can be executed in parallel.</td>
</tr>
<tr>
<td>upgrade step</td>
<td>A collection of pre-defined actions executed by an upgrade procedure in an upgrade campaign. The actions usually consist of deactivating old entities, installation of new software and activating new entities.</td>
</tr>
<tr>
<td>deactivation unit</td>
<td>Contains a group of software entities that must be taken out of service before the upgrade operation.</td>
</tr>
<tr>
<td>activation unit</td>
<td>Contains a group of software entities that are brought into service after an upgrade operation.</td>
</tr>
<tr>
<td>upgrade scope</td>
<td>Defines a set of software entities that are affected by the upgrade campaign. Represented as an union of the deactivation and activation units.</td>
</tr>
<tr>
<td>upgrade method</td>
<td>Defines how an upgrade step should be applied to the deactivation and activation unit pairs in each upgrade procedure.</td>
</tr>
<tr>
<td>upgrade campaign specification</td>
<td>Expresses the actions required for an upgrade campaign.</td>
</tr>
</tbody>
</table>

Figure 5.1 illustrates the major activities of an upgrade campaign without the consideration of failures. After a system administrator has initiated an upgrade campaign, SMF conducts the necessary initialisation steps to ensure that prerequisites such as all the required resources are
available, no other upgrade is ongoing, all essential backups have been created etc. have been met. For each upgrade procedure, actions in each upgrade step are executed and verified. This continues until there are no more upgrade steps to be performed, at which time the results of the upgrade procedure are verified. If there are no more upgrade procedures the results of the upgrade campaign are verified before the whole upgrade operation is committed.

Figure 5.1  Upgrade campaign activities

5.1.2 Upgrade Campaign Specification

All the activities in an upgrade campaign are expressed in an upgrade campaign specification. By generalising and standardising the sequence in an XML schema definition [82], the execution flow of the upgrade campaign can be conveyed in a flexible manner. Figure 5.2 shows the upgrade campaign specification template.
Figure 5.2 Upgrade campaign specification template

An upgrade campaign specification has four sections. In the <campaignInfo> section, general information related to the campaign such as the estimated time required for the upgrade and a reference to the configuration on which the estimate is based are held.
The <campaignInitialization> section contains all the actions needed before the campaign execution, including the prerequisites tests, setting up callback mechanisms for notifying processes that are interested in the upgrade, and any other system specific actions needed.

The <upgradeProcedure> section specifies the core upgrade actions for the campaign are specified. Each upgrade procedure has outage information that states the acceptable level of outage. An expected runtime outage can be calculated by using the deactivation units and the actual deployment configuration. Only when the acceptable outage exceeds the expected runtime outage, the upgrade campaign can be started.

There are two defined methods of upgrade for a campaign. For rolling upgrade, an upgrade step iterates over a set of deactivation and activation unit pairs until the upgrade scope is exhausted. This follows the general principle of rolling upgrade as discussed in section 2.4.3 and illustrated in Figure 2.5 on page 19. In the single-step upgrade method, the upgrade scope is just one big deactivation and activation unit pair, and the upgrade step actions are applied to all the entities simultaneously.

The set of software entities that are affected by the campaign are specified in the upgrade scope, which consists of the deactivation and activation units. The actual actions that apply to the upgrade scope have been generalised and shown in the figure as the five actions. Since the first action of an upgrade step is to take the deactivation unit out of service, the selection of the upgrade scope is therefore a critical means to devise the level of service availability during the upgrade. This can usually be done with the intricate combination of the current configuration information such as how many other software entities are affected if the scope is set to the level of a node or service unit, and the deployed redundancy models. Each of these options has its trade-off in terms of the potential levels of service interruption and protection against failures.

The <campaignWrapup> section contains all the system specific actions needed to verify that the campaign is successful before the final commit of the campaign is carried out.

5.1.3 Failure Handling

Failures may occur during an upgrade campaign and they must be detected and corrected, if possible. Otherwise, service interruption may result. Failure detection is carried out once at the beginning of the campaign by means of a series of prerequisite tests, which include assessments such as whether SMF is operational, if the software repository is accessible, the expected service outage does not exceed the acceptable service outage, and the like. During the campaign, verification of each upgrade step and upgrade procedure is performed before proceeding to the next stage. The whole upgrade campaign is verified before the final commit of all the changes.
The types of failure considered by SMF are those defined by the ITU X.733 alarm reporting function [16]. For example, storage capacity problem, communication protocol error, out of memory, and power problem are some of those 57 defined probable causes. In a SAF system, a specific event known as an alarm report is generated, via the Notification Service [77], whenever such a probable cause is detected by an entity. Being a subscriber to the Notification Service, SMF relies on the propagated error reports to handle failures during an upgrade.

Failure handling involves deploying protective measures in order to facilitate recovery operations if a failure is detected during an upgrade campaign. A full system backup created before the upgrade campaign begins are used to restore the system to the last known correct state if necessary. An update history containing enough information of all the executed actions are also kept for the recovery operations.

![Figure 5.3 Upgrade campaign activities with failure handling](image)

Recovery operations must be performed to undo the effects of the already completed actions. Depending on where the failure is detected, different recovery operations become applicable. Figure 5.3 illustrates the upgrade campaign activities with failure handling, which is
an extension of the previous diagram (shown in Figure 5.1 on page 57) that shows only the successful outcomes of each verification.

If the failure occurs immediately after initialisation and before an upgrade procedure is executed, then there are effectively no changes made into the system. The upgrade campaign can be terminated safely without causing any inconsistency.

If the failure is at the upgrade step level, the step can be undone and retried. The undoing of a failed step is essentially a reverse operation of its forward path counterpart, which can be performed by taking the activation unit out of service, restoring the old software and set off the deactivation unit. If retry is not permitted nor possible, one option is rollback, which reverts the successfully completed steps of the campaign so far by undoing all these steps. After a successful rollback, an explicit commit action is needed to trigger the wrap-up operations of the upgrade campaign. During a rollback, the system continues to provide service.

There are situations in which a rollback is not possible because the system is either in an unknown state, or the time needed for a rollback is longer than the currently available maintenance window. As such, the only option is a fallback, which is the most drastic recovery operation by means of system restart using the image saved during system backup. During a fallback, the system does not provide any service.

If the failure is at the point immediately after the verification of either a procedure or the campaign, rollback and fallback are the available recovery options. A procedure rollback is symmetrical to its forward execution path, during which completed steps are rolled back one by one in reverse order. Similarly, a campaign rollback is symmetrical to the upgrade campaign. The procedures are rolled back in the reverse order of their execution level.

Note that the notions of deactivation and activation units also allow for errors encountered during an upgrade be confined within the corresponding unit, which reduces the complexities of failures handling. The choice of an unit’s size has service availability and performance implications though. For instance, by choosing a very large unit size would simplify the design of an upgrade campaign but the trade-off is the unnecessary interruption of an otherwise unrelated service because of the widening of the upgrade scope. If there are a number of entities on the same node are to be upgraded, the deactivation of an entire node may sometimes have better performance over taking smaller units out of service multiple times.

5.1.4 State Models

In order to deliver coherent behaviour, SMF needs a well defined state model to capture all the normal and failure conditions of a system undergoing an upgrade.

An upgrade campaign is modelled by a set of communicating finite state machines (FSM), as shown in Figure 5.4. For each upgrade campaign, there is an associated FSM. For each upgrade procedure in the same upgrade campaign, there is one corresponding procedure FSM.
For each upgrade step in each upgrade procedure, there is an associated step FSM. A campaign FSM receives signals from the administrator, for example, to execute or suspend a campaign. The campaign FSM then transitions into an appropriate state and communicates the resulting signals to its corresponding procedure FSMs. In turn, each procedure FSM transitions into the appropriate state and communicates with the corresponding upgrade step FSMs accordingly.

The key role of these FSMs is to capture the normal and failure states respectively at the campaign, procedure and step levels such that failure handling can be dealt with in a predictable and consistent way through the use of the upgrade campaign specification XML. A SMF compliant implementation requires that an upgrade campaign specification conforming to the XML schema definition be acted upon according to the defined state models. Therefore, executing the same upgrade campaign specification on another site should produce consistent and predictable behaviour.

All the required states for normal and failure conditions have been defined and standardised in the SMF specification. Readers are referred to Figures 5, 6 and 7 of the standard [78] for the state diagrams of upgrade step, upgrade procedure and upgrade campaign respectively.

![Figure 5.4 Upgrade campaign finite state machines](image)

### 5.1.5 Programming and Administrative Interfaces

SMF provides two types of interfaces, one at the user program level targeting the upgrade-aware applications and the other is for the system administrators.

The programming interfaces allow a user program to be informed about the initiation and progress of an upgrade campaign, after a registration and expression of a scope of interest to SMF. This is particularly useful for applications that require synchronisation with certain
application-specific actions, for example backing up application-specific data. Table 5.2 contains a brief description of the SMF API.

Table 5.2 Software Management Framework API

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>A user process informs SMF that it would like to receive callbacks when the upgrade campaign reaches its registered scope of interests.</td>
</tr>
<tr>
<td>Unregister</td>
<td>A user process informs SMF that it no longer wants to receive callbacks when the upgrade campaign reaches its previously registered scope of interests.</td>
</tr>
<tr>
<td>Response</td>
<td>A user process responds with the result of an SMF initiated operation.</td>
</tr>
</tbody>
</table>

Since an upgrade operation is controlled and monitored by system administrators, a set of interfaces have been defined for this purpose. This includes executing, committing, suspending and rolling back an upgrade campaign, which are explained in Table 5.3.

Table 5.3 Software Management Framework Administrative API

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute</td>
<td>The administrator invokes this operation either to trigger the execution of or resume an suspended upgrade campaign.</td>
</tr>
<tr>
<td>Commit</td>
<td>When an upgrade campaign has completed successfully, the administrator verifies the result of the upgrade campaign and invokes a commit operation.</td>
</tr>
<tr>
<td>Suspend</td>
<td>The administrator suspends the execution of an upgrade campaign, which can be resumed later by the execute operation or rolled back. SMF completes the upgrade steps of one or more upgrade procedures currently being executed.</td>
</tr>
<tr>
<td>Rollback</td>
<td>A rollback operation is performed on a suspended upgrade campaign due to a non-fatal failure.</td>
</tr>
</tbody>
</table>

5.2 Upgrading the Example GGSN

This section presents the use of SMF for upgrading the example GGSN in the previous chapter as shown in Section 4.2 on page 44.

As illustrated in Figure 4.7 on page 47, the example GGSN service is made up of two service units, G1 and G2, which form a service group GGSNservice using the 2N redundancy model. Each of the example GGSN function, namely, session management, PDP address allocation and charging data collection is mapped onto AMF components of types PDPMgt, AddrAlloc and Charging respectively. It is assumed that the instances of component type
Charging (C1, C2) be upgraded by SMF to a new version with no service outage using the rolling upgrade method. It is further assumed that both the old and new versions of the software are compatible with each other.

The first phase of an upgrade is software delivery to the site. A copy of the new version of component type Charging is packed into an entity types file according to the corresponding XML schema definition. Additional information such as compatibility, relative install location and configuration of the new software are also included. This entity types file is then distributed to the software repository in a system dependent manner. The SMF on site updates the status that a new component of type Charging is now available for upgrade.

The second phase of an upgrade is software deployment. This involves the creation of the upgrade campaign XML for accomplishing the task. Focusing on the service availability property, a pseudo upgrade campaign specification is presented in Figure 5.5.

The intended requirement of performing the upgrade without service outage is expressed in the <outageInfo> element as no service instance is allowed to become unassigned during the campaign. If this condition is expected to occur, the campaign will not be even started by SMF.

Given the example GGSN deploys a 2N redundancy model, and that a rolling upgrade method has been chosen, the identified impacted entities for upgrading the Charging components are therefore those associated with service units G1 and G2. Since both service units belong to the same parent entity, which is service group GGSNService, the upgrade method thus has a symmetric upgrade scope. This means that the upgrade procedure has two upgrade steps, and that G1 and G2 form the corresponding deactivation-activation pairs in the actions of each of the upgrade step. The five actions of each upgrade step are then executed sequentially, completing the upgrade of the two Charging component instances while providing service during the process.

In the upgrade campaign specification shown in Figure 5.5, the service group GGSNService is left unprotected whenever the campaign execution is between actions 1 and 5 in each upgrade step. If there is a failure of any entities associated with serving the active service instance of GGSNService, interruption of service is expected. This however is not a problem of the rolling upgrade method per se, instead, it highlights the fundamentally intimate relationship between service outage issues and a system’s currently deployed redundancy model. For instance, by changing to a 2+1 (two active units, one standby) protection scheme the same rolling upgrade method can be applied to the same system, which will be protected even if there is a failure during the upgrade.
5.3 Additional Validation

As pointed out in the Research Method discussion in Section 1.4 on page 7, technical evaluation of SMF has already been carried out as part of the specification creation process. This section provides additional validation by illustrating the level of support offered by SMF for upgrading the example GGSN presented in Section 5.2 above in practice, thus, reinforcing that the SMF solution is solving practically relevant problems.

There are typically three main issues to consider when upgrading a network element product in the field. The first one is the compatibility between the old and new versions of the software, which affects how the actual upgrade should be conducted. The second concerns how the upgrade operation cope with the site-specific configurations of the deployed system, even though the deployed products are the same. The third involves minimising human errors during the upgrade by providing the system administrators with upgrade operations as consistent as
possible, be it the same network element across different sites or different network elements on the same system.

The following illustrates the support provided by SMF for upgrading the example GGSN in the field under various scenarios:

- **Configurable upgrade method.** The concept of upgrade campaign and its associated specification support in XML is a key to providing the necessary mechanism for implementing upgrade method variability. Assuming that the new Charging component software in Section 5.2 above is now incompatible with the old version, a different upgrade method is needed. Nevertheless, the upgrade campaign specification in Figure 5.5 on page 65 can still be reused. With just one change from `rollingUpgrade` to `singleStepUpgrade` in `<upgradeMethod>`, and the upgrade step is reduced from two to one, the same five actions within the previous upgrade step are performed on all the identified software entities simultaneously to achieve the intended upgrade.

- **Site-specific upgrade.** Lower level variations such as those related to system or site-specific actions are supported by the configurable upgrade campaign specification at various stages. For example, in the campaign initialization `<campaignInitialization>` and wrap-up `<campaignWrapup>` phases shown in Figure 5.5 on page 65 there are optional, customisable actions. At the upgrade procedure level `<upgradeProcedure>`, there are hooks for including custom-built initialisation and wrap-up actions. In each upgrade step, there are options for incorporating application callbacks at various predefined points where an action is executed. The standardised upgrade campaign specification therefore provides portability of the main parts of the upgrade description to the same GGSN product at a different site.

- **Consistent service availability concept.** The application independent notion of AMF’s service instances, which forms the basis of the definition of service outage estimation and monitoring in SMF, allows the same notion of service interruption to be used for designing an upgrade campaign for products other than the GGSN example above in a consistent way. Furthermore, the breaking up of an upgrade campaign into upgrade procedures and upgrade steps in SMF facilitates the reuse of these smaller size actions in other upgrade operation on systems with similar configurations, for example, they can be shared for upgrading products in the same product line.

However, the SMF solution has inherently a dependency on AMF. Therefore, there is no total freedom of picking and mixing the availability management middleware and online software upgrade solution.
There are two observed limitations of SMF. Before a piece of software can be upgraded, the new version must first be delivered by a software vendor and then installed on the target system during the upgrade process. This requires the new software be packaged in a standardised manner, containing information such as the types of software being delivered, versioning, dependency on other packages, compatibility information, configuration of the new software and the relative installation location. The *entity types file* is such a descriptor file, the format of which has been defined by an XML schema [81]. However, this definition only supports types recognised by AMF, which include component, component service, service unit, service group, service and application to be upgraded. The more common elements found in carrier-grade platforms such as firmware and operating systems are currently not yet supported.

Since the SMF specification is only in its first release, there are still a number of areas that require further development. Similar experience on the evolution of AMF in the past has indicated that between one to two iterations of the standard are needed in order to feed the crucial practical experience back into the specification. On the pragmatic front, interfacing the software image management to *de facto* standards of package management software such as RPM (Redhat Package Manager) [54] or Debian [24] package management system is deemed necessary. On the more theoretical side, there is not enough understanding of all the intricacies of how the combination of an upgrade scope with a service group redundancy model may impact the overall service availability. A more formal relationship among these entities is needed. It is expected that such further investigation would discover more optimised ways of upgrading through new methods, resulting in delivering more automation to the process.

### 5.4 Related Work

IBM’s Tivoli® Provisioning Manager [35] is a popular product for automating software deployments in large-scale IT environments across different locations. It provides a comprehensive software life cycle management system from distribution, installation, verification to reporting. The essential components in the product are quite similar to those found in SMF. For example, the data model (Information Model) contains the deployment configuration descriptions; the automation package (upgrade campaign specification) has a collection of workflows and scripts that automates the provisioning of software; the compliance and remediation (failure detection and failure handling) management allows for the revealing of noncompliance and the generation of their recommended corrections; the operator and administrator console (administrative API) provides for the monitoring and controlling of the software deployment. However, one crucial difference is that the product does not deal with availability management during software deployment. Service availability therefore is not guaranteed.
Other popular means of upgrade used in standards-based systems are RPM under the Linux Standard Base specifications [54] and Debian software package format [24]. They define the formats which are necessary to resolve compatibility and dependency issues. Together with the corresponding utilities, they handle the distribution and installation of new software. However, they do not address any of the procedural aspects of upgrades.

The Open Group’s Distributed Software Administration (XDSA) [66], which is based on the IEEE 1387.2 standard, goes beyond the issues tackled by RPM. The XDSA specification defines the package layout and addresses the software administration requirements of distributed systems and applications. The specification defines a set of administration utilities that enable management applications to control the packaging, distribution, installation, update and removal of software across multiple nodes of a cluster. XDSA still does not address the service availability issue in any way.

The OSGi Alliance [67] has defined a number of standard specifications for multiple Java™ based components to cooperate in a single Java Virtual Machine. The principle behind the OSGi technology enables applications to be developed from small and reusable components, which are then composed and deployed. The core part of this technology is the OSGi Framework, which consists of a number of layers that include the Execution Environment, Modules, Life Cycle Management, and Service Registry. Specifically, the Life Cycle Management layer enables an application to be dynamically installed, started, stopped, updated, and uninstalled. This is similar to the facilities provided by the combination of SAF’s AMF and SMF. However, the OSGi technology does not explicitly provide support for managing the availability of applications made up of cooperating components.

The Object Management Group’s Online Upgrade specification [62] addresses the upgrade of a CORBA object implementation. Interfaces have been defined for an upgrade manager to prepare an object for upgrading; to perform the upgrades of one or more objects; to rollback upgrades of objects; and to revert an object from its new implementation to its old implementation. The first step of an upgrade is to put both the old and new implementations into an object group, followed by the query to the old implementation to determine whether it is safe to perform the upgrade. A vendor specific implementation of the upgrade mechanisms is expected to stop new messages being delivered to the old implementation, instead, it queues them for delivery to the new implementation. When the old implementation responds affirmatively to the upgrade request, the current state of the old implementation is transferred to the new one. The queued messages are then replayed to the new implementation, and all future messages for this object group are directed to the new implementation. The old implementation is then removed from the object group before the upgrade is committed. If some part of the upgrade fails, an implementation specific rollback and reverting an upgrade could be used to recover. Some central features in the Online Upgrade specification are missing, for example,
object implementations are not versioned and concurrent operation of the old and new implementations is not supported. In addition, it does not really support a true online upgrade because services provided by the object to be upgraded are not available during the upgrade, although the incoming messages for that object are preserved. To overcome this limitation, it has been recommended that the Fault Tolerant CORBA should be used in order to exploit the inherent redundancy for having minimum, or even no loss of service during an upgrade. However, there is no guidelines on how to integrate these two in a seamless manner.

Mirage [[19], a distributed framework that integrates a staged deployment, user-machine testing, and problem reporting into the software upgrade development cycle. Staged deployment involves clustering the machines for the purpose of an upgrade and testing the upgrade on representative machines before moving to the non-representative ones. The deployment protocol defines the potential parallelisms of an upgrade operation. Environmental resources are identified including the operating system, runtime libraries, configuration files etc. that each application uses. A difference is calculated based on the variation between a vendor reference machine and a user machine, which is used as the basis for clustering. User machine testing involves a comparison of the behaviour of an application before and after an upgrade. It checks for dependency, collects the inputs and outputs passed to an upgraded application, and validates an upgrade by comparing the collected input and output data. The reporting subsystem provides feedback that includes information about the cluster of deployment, success or failure results and report images that allow the upgrade problems to be reproduced to the vendor. Mirage does not deal with service availability in the context of cooperating applications. There is no notion of continuous operation defined. It mainly focuses on the preparation before the actual deployment, specifically the testing of how the upgraded application behaves in the live environment. SMF states that these preparations must be performed but no standardisation is needed.

A framework for live software upgrade, in which a dynamic software architecture and communication model allow for software modules to be replaced on the fly has been proposed in [94]. This involves inserting a module proxy, which handles the input request queue, before an module implementation. Just before an upgrade operation, the module proxy collaborates with the module implementation to achieve the quiescent state by completing all the pending input requests. The module proxy then associates itself with the new implementation. An upgrade protocol is used to specify the transition scenario, which is similar to SMF’s upgrade campaign specification. The framework provides mechanisms for maintaining state consistency and uses a two-phase commit protocol to ensure atomicity of upgrade transactions. A command line interface, which is comparable to SMF’s administrative APIs, facilitates reconfiguration management. Although the proposed framework is applicable to multi-task software, it is only catered for a centralised environment.
A design for concurrently supporting multiple versions of software in a cluster, which is critical for rolling upgrade, has been presented in [70]. The assumption is that different versions of the cluster operation system, which is the software to be upgraded, are interoperable. The infrastructure is CORBA-like with the Object Request Broker (ORB) inside the kernel supporting its subsystems, which are structured as CORBA objects. The ORB however is not associated with the general applications. Interface of a software component is specified in Interface Definition Language (IDL) with the versioning information within a module. A new version of software requires a new IDL, and a new translator module is created for the previous version. A translator object implements an old interface and co-locates with the new implementation. A translator object is created and destroyed together with the implementation object. A vertex object de-multiplexes incoming requests to the appropriate handlers, either the new implementation or a translator. The design concentrates on supporting the coexistence of multiple versions as a basis for rolling upgrade a cluster operating system. No service availability is considered during the upgrade, nor its error handling.

The use of Redundant Arrays of Independent Components (RAIC) to provide dependable services, allowing for components be added or removed dynamically through a RAIC controller as one of the many use cases, has been discussed in [55]. Component types are stateless and stateful, and component state recovery is carried out by means of snapshots or call-history when a new component is used to replace a failed one. Just-in-time testing is a means for detecting errors of a component. RAIC defines seven levels (0-6) of redundancy models and three (a-c) kinds of invocation models by the controller, denoting sequential, synchronous parallel and asynchronous parallel. Classification of a system using the RAIC level is a way of describing its properties. In the case of an upgrade, component relations such as interfaces form the basis of integration strategies that determine how the components are used together. The assumption is that applications do not work together to provide a service. An upgrade operation inserts a new component into a RAIC with the same interface, which leads to an application running in a RAIC with different versions. RAIC controllers mask errors during an upgrade by invoking an alternative version if an exception occurs. However, a fundamental limitation is the lack of component dependency information.

A classic paper [22] argues that the spheres of control exist as the logical boundaries in all systems. The principle is application independent and the premise is that loss of control is therefore loss of integrity in a system. The boundaries are delineated by operators, which possess a number of properties. Process control ensures that an appropriate scope is drawn to isolate the parts concerned from the rest, limiting the unnecessary disturbance within a system. Process atomicity, dependency management and execution in serial or parallel mode are some of the mechanisms. Recovery control preserves integrity by providing the required actions to revert a system to a sane state, be it at the point of before or after commitment. Audit control
determines the validity of single actions or a set of related processes that are disjoint in time. Relational integrity control is all about consistency and coherent behaviour within a system. This basic principle can be applied to all kinds of data processing systems including privacy control, transaction control and version control. SMF has implemented all these controls for online software upgrade through upgrade scope for process control; upgrade step undo, rollback and fallback for recovery control; verifications for upgrade step, procedure and campaign for audit control; and the state models of upgrade step, procedure and campaign for relational integrity control.

An architecture for managed upgrade has been advocated in [31] as a solution to address the issue of upgrading composite Web Services online while maintaining certain level of dependability. The approach introduces a specialised middleware that intercepts user requests and the responses of the old and new versions of the Web Services, a monitoring subsystem to assess the confidence of the different versions of the Web Services based on Bayesian inference, and a management subsystem to adjudicate and return the response according to the confidence assessment. Simulation results indicate that the architecture is useful, although there are still future work needed to investigate different ways of notifying consumers about a Web Service has been upgraded and the impact of the imperfect detection issue on predicted confidence. Although both this approach and SMF deal with dependable upgrade on standardised constituent components, the fundamental difference is that SMF performs the online software upgrade in a standardised way. This means that the upgrade operation is portable to other products in the same product line as well as different sites, which is a rather important requirement for a building block.

5.5 Summary

The solution for supporting online software upgrade in carrier-grade platforms has been standardised by the Service Availability Forum as the Software Management Framework specification.

In order to maintain service availability, the upgrade campaign has been developed to structure the upgrade operation into smaller parts as upgrade procedures and upgrade steps for a consistent, coordinated execution. An XML schema has been defined to express the execution flow of an upgrade campaign. There are two aspects in SMF that address the issue of service interruption during an upgrade campaign. The first one is the failure handling capability of SMF while the second is built into the upgrade campaign specification. The level of service availability during an upgrade campaign execution can be expressed in an upgrade campaign specification, for example, the outage information entry states the acceptable level of service disruption. In addition, the upgrade scope is a critical means to adjust the service availability
level by exploiting the different membership of the deactivation and activation units according to the current configuration as well as the deployed redundancy model.

The state models of upgrade step, upgrade procedure and upgrade campaign have been defined and standardised so that executing the same upgrade campaign specification on another compliant system should produce consistent and predictable behaviour.

For the upgrade-aware applications requiring synchronisation with certain application-specific actions during an upgrade, the programming interface allows for a user program to be informed about the initiation and progress of an upgrade campaign, after a registration and expression of a scope of interest to SMF. Since an upgrade operation is controlled and monitored by system administrators, a set of interfaces have been defined for executing, committing, suspending and rolling back an upgrade campaign.

As an additional validation of the SMF solution, a case study of upgrading the previously shown example GGSN service using the rolling upgrade method is presented. The additional evaluation has highlighted that SMF is indeed applicable in practice. For example, compatibility issues between old and new versions, site-specific upgrade requirements, and provisioning of consistent operations to minimise human errors have all been incorporated into the SMF solution.

Although all the reviewed works have considered aspects of online software upgrade, SMF is the only one that has comprehensively considered upgrade for a distributed application with no or minimal service interruption. In addition, SMF has been shown to have implemented the required spheres of control for online software upgrade.
6 Selecting a Standards-Based Availability Management Middleware

This chapter discusses the support required for selecting a Commercial Off-The-Shelf (COTS) standards-based availability management middleware that is used as a building block in a carrier-grade platform for network elements. The discussion explains the role of conformance testing and robustness testing in the assessment and selection process, describes the testing approaches taken and gives an analysis of the experimental results. An outline of how dependability benchmarking may be integrated into a software product line of network elements is presented. This chapter is an extension of included publications [P5], [P6] and [P7].

6.1 Motivation

Deploying a standards-based system can be characterised as first of all choosing an appropriate standard for the application domain, followed by evaluating a list of conforming candidate implementations and selecting one to be integrated with the proprietary portion of the final system. The decision for the final choice depends on numerous factors such as its level of conformance to the mandatory and optional requirements of the chosen standard, performance, robustness, costs, interoperability, integration with legacy systems, technical support, and additional capabilities.

![Figure 6.1 Network element life cycle](image)
Figure 6.1 illustrates a simplified life cycle, which consists of the development and use phases, of a network element utilising a standards-based availability management middleware in the carrier-grade platform. The development phase is divided into the standards-based service availability middleware and proprietary components tracks, shown in the upper and lower parts of the diagram respectively. The standards-based service availability middleware track involves an assessment and selection of suitable COTS SAF service availability middleware implementations, and subsequently an integration of the selected candidates into the proprietary part of the network element. The proprietary components include those in the carrier-grade platform and the application. In the use phase, the diagram shows that both the selected AMF and SMF implementations are part of the network element, together with software entities that are managed by the AMF as SA-aware components, and those that are independent of AMF as non SA-aware components. This phase also deals with the maintenance of a network element itself and therefore SMF is shown in the figure to support this through online software upgrade.

The assessment and selection of a suitable service availability middleware in the development phase along the standards-based service availability middleware track is crucial. The assessment of a COTS service availability middleware must be conducted on its functional and non-functional properties. The former focuses on whether a candidate delivers the functions as specified in the SAF service availability standards, while the latter concentrates on the quality of its implementation. Due to the need of the subsequent technical support on the middleware, managing the risks associated with the stability of the supplier company is equally important.

The COTS service availability middleware selection decision has a long-term effect because a recognition of picking a sub-optimal candidate later on in the life cycle would cost considerable resources and effort to correct. This requires that the decision made be objective, and the process be repeatable. The objectivity is typically backed by measurement-based data while a repeatable process is usually delivered by adopting best practices in the industry.

One of the most common ways of assessing conformance to some specifications is by means of testing. It aims at determining whether a prospective standards-compliant product meets the specified standard by carrying out certain test procedures, which involve performing many test cases and making observations that the system under test exhibits the defined functional properties. In terms of assessing whether a COTS service availability middleware is indeed SAF compliant, some kinds of conformance testing are needed.

Out of all the non-functional properties of a COTS service availability middleware, robustness is evidently the most important to consider in carrier-grade systems. The side effect of crashing or hanging of a third-party service availability middleware in such a system causes the whole network element to fail as well. In practice though, when a product is chosen from a pool of suppliers robustness is very often over shadowed by performance. The “seems to work
well” criterion is typically the most commonly used without any further consideration of a product’s robustness. The challenge here is to develop the appropriate technology for measuring and comparing robustness of COTS SAF service availability middleware in order to support the assessment and selection process.

The following sections discuss the approaches and experimental results of conformance testing and robustness testing for SAF AMF implementations.

6.2 Conformance Testing

Conformance testing is normally undertaken by an independent organisation in a certification process and a logo signifies that the product bearing the mark does conform to the specified standard. In the absence of a certification programme in place, such as the case in the Service Availability Forum at the time of writing, conformance assessment is therefore necessary in the SAF middleware assessment and selection sub-phase. In terms of an interface specification standard such as AMF, this means that a compliant product must implement all the defined AMF APIs according to each function’s syntax and semantics in the specification. Typical issues of conformance testing are to find a way of efficiently generating the necessary test cases and ensuring that they have sufficient coverage.

6.2.1 The IEEE 2003-1997 Approach

In order to ensure that the conformance assessment process is repeatable, a well defined procedure must be followed. The requirements and guidelines offered by the IEEE standard 2003-1997 [37] for measuring conformance to the POSIX® standards is considered to be applicable, primarily due to the similarity that the AMF APIs have also been defined as a collection of C functions. Table 6.1 contains the main definitions of this standard.

Figure 6.2 illustrates the basic model for conformance assessment as defined by the IEEE 2003-1997 standard. The assessment process can be broken up into three stages. The first stage is to develop the test method specification (TMS) by following the guidelines in IEEE 2003-1997 on the base standard, which in this context is the AMF specification of the Service Availability Forum’s AIS. The TMS contains a number of assertions written for each element, which corresponds to one AMF function as defined by its APIs. Each assertion identifies what is to be tested and must be stated in such a way that a test result code of PASS indicates conformance to AMF. By mapping every condition associated with a defined return code in each AMF function onto one assertion ensures that all the specified behaviours are covered, which include the successful and anticipated error invocations. Further, a set of conforming test result codes is stated for each assertion that a test program can report for a conforming
implementation. The TMS is mainly expressed using an informal, natural language such as English.

**Table 6.1 IEEE 2003-1997 definitions**

<table>
<thead>
<tr>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>assertion</td>
<td>A definition of what is to be tested and the outcome is TRUE for a conforming implementation.</td>
</tr>
<tr>
<td>base Standard</td>
<td>The standard for which the conformance assessment is sought.</td>
</tr>
<tr>
<td>conforming implementation</td>
<td>An implementation that satisfies all the conformance requirements.</td>
</tr>
<tr>
<td>conformance test software</td>
<td>Test software used to ascertain conformance to the base standard.</td>
</tr>
<tr>
<td>element</td>
<td>A functional interface in the base standard.</td>
</tr>
<tr>
<td>IUT</td>
<td>Implementation Under Test.</td>
</tr>
<tr>
<td>test method implementation</td>
<td>The means, typically a set of software and procedures, used to measure the conformance of an IUT to a base standard.</td>
</tr>
<tr>
<td>test method specification</td>
<td>A document that expresses the required functionality and behaviour of a base standard. These are described as assertions, together with a complete set of conforming test result codes.</td>
</tr>
<tr>
<td>test result code</td>
<td>A value that indicates an assertion’s test result.</td>
</tr>
</tbody>
</table>

**Figure 6.2 Basic model for conformance assessment**
The second stage includes developing the test method implementation (TMI) and conducting the conformance testing. A test method implementation developer takes each assertion in the TMS and turns it into one or more tests in the conformance test software. The TMI is documented with information such as how to install, configure and execute the test software, gather and interpret results, and known limitations. During the execution of the test software, intermediate result codes may be produced to provide more information on why a specific test result is issued. A final test result code is then determined for an assertion test.

The third stage deals with the assessment and reporting of the conformance test. A test report containing information including the identifications of the base standard, TMS and TMI, results of each assertion test, and the date of the test is produced. An implementation under test (IUT) is judged to be conforming if all the final test result codes match with those conforming test result codes in the TMS. For the non-conforming implementations, the test report is the source for identifying the functions and the likely reasons of failures.

The IEEE 2003-1997 standard distinguishes three levels of conformance testing. The first option is based on exhaustive testing, which seeks to verify the behaviour of every aspect of an element in all conceivable permutations. This however is normally infeasible due to the excessive number of tests required, not to mention the multiple combinations of the deployment hardware configurations.

At the other end of the spectrum is the second option known as identification testing whereby only a cursory examination is required. In this case, simply having the C function prototypes that match with the defined AMF APIs is considered to be conforming. This approach clearly lacks the confidence sought in an implementation claiming to be conforming and is therefore considered not useful.

The third alternative is called thorough testing in which the behaviour of every aspect of an element is verified in isolation but it does not require the coverage of all possible permutations. This option offers the possibility of reducing the number of required tests. In terms of testing an AMF API function, this means that a function can be tested with making one parameter invalid at a time. The number of test cases required is therefore the number of parameters plus the one case with all valid parameters.

6.2.2 Experimental Results and Evaluation

The included publication [P5] reported the experience of thorough testing an internally developed AMF implementation. This experiment of applying the IEEE 2003-1997 guidelines was carried out at the time when SAF member companies were investigating into ways of conducting the certification process for products claiming compliance. Further, the base standard was the first release of the AMF specification [75].
The following is an example of a TMS for the AMF get component capability model function. Figure 6.3 shows the definition of the function.

The function takes a component name as input and returns its associated component capability model. An error code of type `SaErrorT` is returned, which is an enumerated type definition of globally unique constants representing all the specified error codes for all the AMF functions.

```c
typedef enum {
  SA_AMF_COMPONENT_CAPABILITY_X_ACTIVE_AND_Y_STANDBY = 1,
  SA_AMF_COMPONENT_CAPABILITY_X_ACTIVE_OR_Y_STANDBY = 2,
  SA_AMF_COMPONENT_CAPABILITY_1_ACTIVE_X_OR_Y_STANDBY = 3,
  SA_AMF_COMPONENT_CAPABILITY_1_ACTIVE_OR_1_STANDBY = 4,
  SA_AMF_COMPONENT_CAPABILITY_X_ACTIVE = 5,
  SA_AMF_COMPONENT_CAPABILITY_1_ACTIVE = 6,
  SA_AMF_COMPONENT_CAPABILITY_NO_STATE = 7
} SaAmfComponentCapabilityModelT;

typedef unsigned short SaUint16T;

#define SA_MAX_NAME_LENGTH 256

typedef struct {
  SaUint16T length;
  unsigned char value[SA_MAX_NAME_LENGTH];
} SaNameT;

SaErrorT saAmfComponentCapabilityModelGet(
  const SaNameT *compName,
  const SaAmfComponentCapabilityModelT *componentCapabilityModel
);
```

**Figure 6.3 Get component capability model function**

Limiting to two assertions in this example, Table 6.2 shows the entries in the TMS in which a successful and a failed invocation of the function. For each assertion, the setup, how the test should be conducted, and the expected error code are described.

**Table 6.2 Example assertions**

<table>
<thead>
<tr>
<th>Assertion</th>
<th>Setup</th>
<th>Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>ok</td>
<td>Use a previously registered component in <code>compName</code>.</td>
<td>A call to <code>saAmfComponentCapabilityModelGet()</code> returns the component capability model used by the component identified by <code>compName</code>. The call returns <code>SA_OK</code>.</td>
</tr>
<tr>
<td>error6</td>
<td>Use a component that is not previously registered in <code>compName</code>.</td>
<td>A call to <code>saAmfComponentCapabilityModelGet()</code> returns <code>SA_ERR_NOT_EXIST</code>.</td>
</tr>
</tbody>
</table>
Table 6.3 shows a comparison of the number of required test cases for exhaustive and thorough testing. The names, number of error codes and parameters for each AMF function are listed in the first three columns.

<table>
<thead>
<tr>
<th>AMF functions</th>
<th>Error Codes</th>
<th>Parameters</th>
<th>Exhaustive test cases</th>
<th>Thorough test cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>saAmfComponentRegister</td>
<td>11</td>
<td>3</td>
<td>88</td>
<td>44</td>
</tr>
<tr>
<td>saAmfComponentUnregister</td>
<td>9</td>
<td>3</td>
<td>72</td>
<td>36</td>
</tr>
<tr>
<td>saAmfCompNameGet</td>
<td>8</td>
<td>2</td>
<td>32</td>
<td>24</td>
</tr>
<tr>
<td>saAmfReadinessStateGet</td>
<td>8</td>
<td>2</td>
<td>32</td>
<td>24</td>
</tr>
<tr>
<td>saAmfStoppingComplete</td>
<td>7</td>
<td>2</td>
<td>28</td>
<td>21</td>
</tr>
<tr>
<td>saAmfHASStateGet</td>
<td>9</td>
<td>3</td>
<td>72</td>
<td>36</td>
</tr>
<tr>
<td>saAmfProtectionGroupTrackStart</td>
<td>8</td>
<td>5</td>
<td>256</td>
<td>48</td>
</tr>
<tr>
<td>saAmfProtectionGroupTrackStop</td>
<td>6</td>
<td>2</td>
<td>24</td>
<td>18</td>
</tr>
<tr>
<td>saAmfErrorReport</td>
<td>8</td>
<td>5</td>
<td>256</td>
<td>48</td>
</tr>
<tr>
<td>saAmfErrorCancelAll</td>
<td>8</td>
<td>1</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>saAmfComponentCapabilityModelGet</td>
<td>7</td>
<td>2</td>
<td>28</td>
<td>21</td>
</tr>
<tr>
<td>saAmfPendingOperationGet</td>
<td>8</td>
<td>2</td>
<td>32</td>
<td>24</td>
</tr>
<tr>
<td>saAmfResponse</td>
<td>6</td>
<td>2</td>
<td>24</td>
<td>18</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>13</strong></td>
<td><strong>103</strong></td>
<td><strong>960</strong></td>
<td><strong>378</strong></td>
</tr>
</tbody>
</table>

Let $EC$ be the number of error codes, $P$ be the number of parameters for each function:

Number of cases required for exhaustive testing $= EC \times 2^P$

Number of cases required for thorough testing $= EC \times (P + 1)$

The number of error codes defined in a function is also the number of assertions in the TMS. For exhaustive testing, the number of test cases required for each assertion is the number of permutations of all its valid and invalid parameters ($2^P$). The strategy adopted for thorough testing is to go through one parameter at a time by making it invalid as one case. The number of test cases required is therefore the number of parameters plus the one case with all valid parameters.
parameters \((P + 1)\). By excluding all the permutations of return codes, the number of tests required is reduced considerably.

The technique has been successfully used for conformance testing an internally developed AMF implementation, which supported seven AMF functions. The TMS contained 55 assertions and 31 test cases were implemented. A test report recording all the verdicts of the conformance test for each supported API was produced. A review meeting was conducted with the team of IUT developers to evaluate the conformance testing process and assess the suitability of the chosen thorough testing strategy.

The conclusion of the review has confirmed that following the requirements and guidelines of the IEEE 2003-1997 standard was indeed feasible and repeatable. The thorough level of testing has the realistic balance of confidence and manageability. During the review of the test results with the IUT developers, no falsely passed tests were identified that would have indicated inadequate coverage of the test cases. On the contrary, one unexpected outcome was that the intermediate test results provided some useful insight into the future implementations for the developers.

In addition, clarifications and feedback arising from this experimentation were reported back to SAF. As a result, they were incorporated into subsequent releases of the AMF specification. As a precursor to the SAF certification programme to include an independent testing facility to certify compliant products, the SAF Test open source project [86] has since adopted this approach to develop a set of conformance test suites.

Focusing on the functional properties of an implementation, conformance testing does not provide any indication of the quality such as robustness of a candidate SAF middleware. The following sections in this chapter explore this subject.

6.3 Robustness Testing

This section explains what is robustness testing and the motivation for using it to select a COTS AMF middleware. It describes the approach adapted for AMF and shows how it is used in the COTS selection process by comparing three candidate implementations. It gives an evaluation and discusses the potential extension into dependability benchmarking that is applicable for software product lines.

6.3.1 Adaptation to AMF

Robustness is a secondary attribute of dependability. It is defined in [36] as the degree to which a system operates correctly even in the presence of exceptional inputs or stressful environmental conditions. Accordingly, robustness faults are those faults that can be activated by these inputs and conditions, resulting in an incorrect operation of a system.
Robustness testing aims to determine a system’s quality of being able to cope well with variations in its operating environment without losing its functionality. By exercising the system under test with a range of exceptional inputs and conditions, observations are made in order to establish the level of robustness. This is rather similar to functional black box testing except that it concentrates on the potential robustness faults. Software that is put through robustness testing tends to be mission critical, or requires assurance from the tests that the software continues to operate normally even after a simulated long period of time. Operating systems and middleware very often undergo this test as both are eventually used by other software that may not even exist at the time, and thus it may not be possible to establish in advance all the different ways in which they will be used. Since it is essential that a network element must have a high level of service availability and long mission time, testing the robustness of the COTS service availability middleware before integrating it into a carrier-grade platform is pertinent.

Robustness testing is a time and resource consuming activity. Generating an effective test suite, executing it and evaluating the results usually needs a lot of manual work. The starting point of testing robustness of AMF implementations is the interface specification, where the number of robustness faults per function is measured. Since the AMF is defined as an interface specification, generating the test cases directly from the API functions and subsequently automating the test execution are possible.

The first step of defining the testing strategy for AMF middleware is to identify the possible sources of inputs that can activate robustness faults. Exceptional inputs can be grouped into the following categories:

- Syntactically incorrect values, e.g. invalid string for an IPv4 address.
- Semantically incorrect values, e.g. non-existing version number.
- Values used in invalid context, e.g. uninitialized handle.

A simple approach uses a set of generic values for all the parameters which are one of the basic types in all the functions as exceptional inputs. However, most of the types used in the AMF functions are defined as complex structures. Using all possible combinations in this set of generic values may result in far too many test cases, especially when AMF functions have on average two to three parameters.

A more structured approach, known as type-specific input, exploits the parameter types in each AMF function by producing test programs with unique exceptional input values according to each parameter type. The principle is to establish a type hierarchy in which cases of a child type inherit all the exceptional input value combinations from their ancestor cases. Instead of defining the test cases one by one for each AMF function, the Template-Based Type-Specific Test Generator (TBTS-TG) tool has been developed to automate this process.

Figure 6.4 shows the input and output flow of the TBTS-TG tool. Central to the TBTS-TG tool are the type-specific test value generator and test generator. The former constructs the C
code for the test value generators in the test sources based on the type metadata of each parameter, the exceptional test values defined as C code snippets and the C skeleton of the generator. The latter is an XSL (eXtensible Stylesheet Language) transformation that uses the function metadata and test case templates to be populated with test values to produce the test case sources. The test case sources and test value generators are compiled and linked with the common functions to become a test case, which is then executed with the result logged after completion. A test case is considered to detect a robustness failure if the test program or the middleware implementation crashes or hangs, due to a segmentation fault or a timeout for example. Although a finer grain classification of the robustness failure types could have been possible, they do not provide any further value from the COTS selection perspective, as opposed to the development standpoint.

![Figure 6.4 Template-Based Type-Specific Test Generator](image)

6.3.2 Experimental Results and Evaluation

By means of running the generated robustness testing cases with the generic and type-specific input values on OpenAIS version 0.69 [65], the relative complexity and effectiveness of the two approaches can be evaluated. The objective was to understand the relative merit of using these two techniques in robustness testing, instead of testing the then current OpenAIS implementation. A validation of the robustness testing approach has been carried out by comparing the robustness of three COTS AMF implementations.

In the generic and type-specific input values comparison, two sets of input were used under generic input. The first one was \{0, -1, 1, fixed random\}. The value 0 is a common input since it represents a NULL when cast to a pointer. The values 1 and -1 are useful for integer or float types, although for pointers they are cast to memory addresses usually reserved for the system and cause a segmentation fault when de-referenced. However, it resulted in robustness failures
that could not be traced back to the individual parameter values that activated the failure, due to all the values used in the function calls were potentially exceptional ones. The second set \{0, valid address\} was used specifically to determine which functions failed to implement null value checks.

Table 6.4 Number of lines of source codes

<table>
<thead>
<tr>
<th>Technique</th>
<th>Test template</th>
<th>Transformations</th>
<th>Metadata</th>
<th>Test values</th>
<th>Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generic</td>
<td>120</td>
<td>80</td>
<td>417</td>
<td>1</td>
<td>618</td>
</tr>
<tr>
<td>Type-specific</td>
<td>323</td>
<td>690</td>
<td>726</td>
<td>254</td>
<td>1993</td>
</tr>
</tbody>
</table>

Table 6.4 shows the number of lines of source code for a function using the two techniques. The initial version of the generic input was created in approximately three days, while the implementation of the framework of type-specific testing required about two weeks. The main advantage of the automated testing approach is that the type-specific testing of a new function requires only the completion of the metadata, and supplying the test values and logging code for the new types used in the function.

Table 6.5 Number of faults by functions

<table>
<thead>
<tr>
<th>Function name</th>
<th>Faults</th>
</tr>
</thead>
<tbody>
<tr>
<td>saAmfCompNameGet</td>
<td>1 + 0</td>
</tr>
<tr>
<td>saAmfComponentCapabilityModelGet</td>
<td>1 + 0</td>
</tr>
<tr>
<td>saAmfComponentRegister</td>
<td>2 + 0</td>
</tr>
<tr>
<td>saAmfComponentUnregister</td>
<td>2 + 0</td>
</tr>
<tr>
<td>saAmfDispatch</td>
<td>1 + 0</td>
</tr>
<tr>
<td>saAmfErrorCancelAll</td>
<td>1 + 0</td>
</tr>
<tr>
<td>saAmfErrorReport</td>
<td>3 + 0</td>
</tr>
<tr>
<td>saAmfFinalize</td>
<td>1 + 0</td>
</tr>
<tr>
<td>saAmfHASTateGet</td>
<td>2 + 0</td>
</tr>
<tr>
<td>saAmfInitialize</td>
<td>0 + 2</td>
</tr>
<tr>
<td>saAmfPendingOperationGet</td>
<td>1 + 0</td>
</tr>
<tr>
<td>saAmfProtectionGroupTrackStart</td>
<td>2 + 0</td>
</tr>
<tr>
<td>saAmfProtectionGroupTrackStop</td>
<td>2 + 0</td>
</tr>
<tr>
<td>saAmfReadinessStateGet</td>
<td>1 + 1</td>
</tr>
<tr>
<td>saAmfResponse</td>
<td>1*</td>
</tr>
<tr>
<td>saAmfSelectionObjectGet</td>
<td>1 + 1</td>
</tr>
<tr>
<td>saAmfStoppingComplete</td>
<td>1*</td>
</tr>
</tbody>
</table>

Table 6.5 shows the number of robustness faults uncovered by the two techniques per AMF function. The notation \((x + y)\) used under the faults column indicates that generic input values have found \(x\) faults while an additional \(y\) faults have been discovered by the type-
specific input technique. A star denotes a critical error has occurred during the experiment, which caused a segmentation fault in the middleware executive.

As reported in the included publication [P7], validation of the template-based type-specific testing has been performed by running the generated test cases on three versions of Service Availability Forum’s AMF implementations. They were a stable release (version 0.80.1) and the then latest development trunk of OpenAIS [65], and Fujitsu-Siemens Computers’ SAFE4TRY [28]. This gives a cross-section of code bases for a stable and under development open source project, and production quality implementation.

Table 6.6 Results of template-based type-specific testing

<table>
<thead>
<tr>
<th>Result</th>
<th>openais-0.80.1</th>
<th>openais-trunk</th>
<th>SAFE4TRY</th>
</tr>
</thead>
<tbody>
<tr>
<td>success</td>
<td>24568</td>
<td>26019</td>
<td>29663</td>
</tr>
<tr>
<td>segmentation fault</td>
<td>1110</td>
<td>1468</td>
<td>0</td>
</tr>
<tr>
<td>timeout</td>
<td>467</td>
<td>2178</td>
<td>2</td>
</tr>
<tr>
<td>Total number of calls</td>
<td>26145</td>
<td>29665</td>
<td>29665</td>
</tr>
</tbody>
</table>

Table 6.6 shows the results of testing the robustness of the three systems by using the template-based type-specific input technique. Note that the total number of calls listed for openais-0.80.1 is less than the other two because for one function (saAmfProtectionGroupTrackStart), the test program and the middleware crashed at the beginning and therefore no subsequent calls were executed. The measured results have indeed supported the validation of the test cases in that the more mature a product is, the less defects it contains.

The test suites and the associated test results for the two OpenAIS implementations have since been released to the open source community [65] as feedback.

6.3.3 Extensions

As a result of the investigative studies of robustness testing in the selection phase of a standards-based COTS availability management middleware for carrier-grade platforms, two extensions have been identified. The first is to enhance the robustness testing by incorporating sequencing support for testing certain state-based AMF functions. The second is to embrace dependability benchmarking and turn the robustness test suites into reusable assets for supporting a software product line of network elements.

A state-based AMF function requires a specific call sequence by an application in order for the middleware to reach a state where the service can be provided. An example is the component response to framework requests function (saAmfResponse). The response can
correspond to one of the nine types of requests such as healthcheck, terminating a component etc. The exceptional condition for such a function could be that a request had never been sent. More advanced robustness testing, including additional test cases, must be able to exercise these conditions.

A mutation-based testing has been experimented as a potential strategy for improving fault detection in more complex scenarios, which usually involve the accumulation of AMF state information during a series of calls by an application. The basic idea of mutation-based testing is that mutation operators representing typical robustness faults such as omitting a call or changing the specific order of calls, are applied to valid functional test programs that exercise the AMF. Five mutation operators have been implemented, they are omission, relocation and swapping of calls, modifying conditions, and replacing parameters.

Although the preliminary results in included publication [P7] have highlighted additional robustness failures that were not detected by the type-specific tests, further work is still needed to validate this approach, as well as having the usefulness of this strategy analysed. Further fundamental work is also needed to understand the relationship between mutation operators and their corresponding call sequences, in order for the resulting coverage determined.

The second extension is to exploit dependability benchmarking in the selection and assessment phase of a COTS availability management middleware for carrier-grade platforms. The goal of benchmarking the dependability of a system is to provide a reproducible way for characterising its behaviour in the presence of faults. The robustness tests on an AMF implementation can therefore be used as a means of benchmarking the resilience to exceptional and stressful conditions of different candidates in a consistent manner.

Figure 6.5 depicts a vision of utilising dependability benchmarking in assessing COTS availability management middleware for carrier-grade platforms. The principle is to have the dependability benchmarking machine running a set of benchmark suite of robustness test cases on a number of COTS availability management middleware products. A fault-load, which is characterised by the type of fault and its rate of activation, can be injected into the dependability benchmarking machine to simulate the exceptional environment. A workload simulating the amount of work to be carried out by the network element is included in the robustness tests execution. The eventual benchmark results for each vendor’s implementation can then be compared based on the same execution condition.

By allowing for such a fault-load and workload pair to be configurable, this principle of operation could be extended to support a software product line in which each product defines its own fault-load and workload pair according to the requirements exhibited in the specific network element application that runs on the platform. The robustness test cases will therefore become reusable assets, which belong to domain engineering and support the notion of
commonality in this context. The configurable fault-load and workload pairs will be the mechanism for providing testing variations in application engineering.

![Figure 6.5 Vision of dependability benchmarking](image)

Dependability benchmarking can also be used to ensure that any higher versions replacement AMF is as robust as the current version, if not better, before an upgrade is deployed in the carrier-grade platform.

### 6.4 Related Work

The issue of COTS product selection is not new, even a standard [40] has been defined for this purpose. However, much of the reported work tend to focus on the process perspective. How the evaluation measurement is performed is left as an application dependent issue.

Procurement-Oriented Requirements Engineering (PORE), a template-based method for acquiring COTS software selection requirements, has been described in [56]. PORE supports iterative requirements acquisition and product selection through the application of three templates in each stage of the process, which deals with the acquisition of the necessary customer requirements and product information to select or reject products. The emphasis is on the guidance of choosing and using each technique, drawn from several disciplines including knowledge engineering, feature analysis, multicriteria decision making, and design rationale. Template 1 uses supplier-given information, template 2 focuses on product demonstration and template 3 gives attention to customer-led hands-on evaluation, reflecting on the requirements acquisition, product modelling and product selection stages. While the first two templates are fairly complete with prescriptive guidance on the techniques to be used, the third one has yet to be developed.
COTS Acquisition Process (CAP), which is a measurement-based COTS assessment and selection method, has been presented in [64]. CAP focuses on the COTS assessment and selection phase, and assumes that full system requirements exist prior to the selection process. The CAP method is made up of three components. Initialization Component identifies the evaluation criteria; estimates the efforts needed to apply all evaluation criteria; sets up measurement plan to conduct the evaluation activities; review step to verify all the IC activities have been performed correctly. The Execution Component has an activity cluster comprising all data collection, evaluation and decision-making steps. Reuse Component packages and stores all useful information generated such as measurement data to decrease the cost of future COTS software acquisition. This method is considered to be objective, reliable, traceable, repeatable, explicit and efficient. Again, this is a process and the mechanism for data collection is not defined.

A hybrid methodology integrating formalised knowledge with human expertise that customises the COTS selection process based on the project domain characteristics has been proposed in [58]. The goal is to provide more informed, transparent and effective decisions than those reached in an ad hoc manner. The conceptual framework involves a three-phase approach. In phase one, the required data for customising the selection process including project domain characteristics, selection activity interdependency and inclusion or exclusion of evaluation criteria are modelled. Phase two explores the different options through the computerised decision support system to refine the options accordingly in order to reach a set of recommendations. Human intervention in phase three to consolidate the results, and if necessary, iterate the process with modified selection methods and pass the control back to phase one for another cycle.

The Ballista [43] testing methodology was a pioneer in robustness testing of COTS components such as an operating system. The basic principle is to generate a set of exceptional parameter values to be used in invoking the functions and can be considered as a form of software-based fault injection. After the tests, the results are classified according to the outcome, including the severity of failure if it happens. It was first shown to compare the robustness of 15 POSIX operating systems from 10 different vendors. It has been further used in [83] for testing six Windows variants of the Microsoft Win32 APIs to obtain quantitative data about its robustness. Similar tests were conducted on RedHat Linux as a foundation for comparing dependability across heterogeneous platforms, which is based on the functional groupings of results as the comparison of similar capabilities but different interfaces. Ballista has also been applied to other software components such as the CORBA middleware [68], in order to test the robustness of the C++ client side exception handling capabilities on two major versions of three ORB implementations over two operating systems. Although it has a similar goal of choosing COTS middleware using robustness testing, the fundamental difference in the
architecture of CORBA and AMF has necessarily affected the immediate use of the test results in the selection process. The former is an incomplete assessment without the servers side whereas the robustness testing in this thesis covers the whole AMF implementation.

DBench [23] was a three-year research project on dependability benchmarking funded by the European Commission under the Information Society Technologies Fifth Framework Programme. The project has developed a framework and guidelines for defining dependability benchmarks for COTS software components, which are used to characterise system behaviours under typical load and common fault conditions. The specific application areas covered were automotive control systems, onboard space control and enterprise systems. The dependability benchmarking proposal in the thesis is an adaptation of the DBench framework, which attempts to integrate complementary solutions of robustness testing and extend them with methods specific to service availability middleware. The incorporation of the product-specific workload and fault-load pairs into the overall benchmarking framework aims at expanding the DBench guidelines for the mobile networks domain.

The desirable features of robustness benchmarks including portability, coverage, extensibility, hierarchy of details, and the reported results are repeatable and representative of the severity of the failure has been presented in [60]. It proposes a hierarchical structured approach to building robustness benchmarks primarily for Unix-like operating systems. While many of these features have been incorporated into the designing of the robustness testing suites in this thesis, the finer grain classification of failures are considered to be less useful when the test results are only used as the basis for selecting a COTS product or not, instead of its further development.

The R-cubed (R³) framework [95] introduces a comprehensive, hierarchical approach to benchmarking system availability. It defines the three key attributes of fault/maintenance rate, robustness and the speed of recovery following an outage, and establishes their corresponding metrics. While the Outage Source Index and Outage Resilience Index metrics under the rate and robustness attributes are important for an overall system availability evaluation, the robustness measures in this thesis are deemed sufficient for supporting the process of selecting an appropriate COTS service availability middleware. The Outage Duration Index metric under the recovery attribute could potentially be exploited in the thesis’ future development. Due to the hierarchal structure of the R³ framework, the speed of recovery could be measured at different levels and granularity, allowing for significant events such as failover, switchover and software upgrade to be incorporated into the benchmarks and used as part of the COTS service availability middleware selection and assessment criteria.
6.5 Summary

Deploying a standards-based system requires an assessment of candidate implementations be integrated into the traditional development process. The selection must be objective and the decision process be repeatable. Assessing whether a candidate product meets a specified standard is usually carried out by performing many test cases and making observations that the system under test exhibits the defined functional properties. For a service availability standard, the non functional property of robustness in a candidate implementation is the most important to consider.

By adopting the requirements and guidelines of the IEEE 2003-1997 standard to carry out conformance testing on an internally developed AMF implementation, it has been concluded that the approach was feasible and repeatable. In addition, the thorough level of testing has the realistic balance of confidence and manageability.

Testing the robustness, that is, quality of being able to cope well with variations in its operating environment, of the COTS availability management middleware before integrating it into a carrier-grade platform is pertinent as a network element must have a high level of service availability and long mission time. The relative complexity and effectiveness of generic and type-specific input values for robustness test cases have been evaluated by experiments. The comparison of three AMF implementations have validated the generated robustness test cases.

The robustness tests on an AMF implementation can be used as a means of benchmarking the resilience to exceptional and stressful conditions of different candidates in a consistent manner. The robustness test cases will therefore become reusable assets, which belong to domain engineering and support the notion of commonality in this context. By allowing for the fault-load and workload pair to be configurable, a software product line of network elements can be supported by having each product defining its own fault-load and workload requirements in the specific network element application that runs on the platform.
7 Introduction to the Included Publications

This chapter summarises the included publications and presents the author’s contributions in each of them. It also includes the author’s contributions to Service Availability Forum’s Availability Management Framework and Software Management Framework specifications.

7.1 On the Development of Standards Based Carrier Grade Platforms

The first attempt of relating the various research conducted in this thesis under a uniform framework for discussion has been presented in [88]. It explains the motivation behind network equipment providers’ strategy of buying instead of building constituent components when developing network infrastructure products. The carrier-grade base platform is introduced as a vehicle for supporting higher level network element applications. It articulate the needs for following standards as a means to reduce development costs. A simplified product life cycle that comprises specification, design, production and operation phases is used to structure the discussions around the impact of adopting a standardised availability management middleware in carrier-grade platforms. The paper proposes the use of a standardised availability management middleware and argues that it must be reflected in the architecture in the design stage. Dependability benchmarking is considered to be needed in the production stage for selecting an off-the-shelf availability management middleware, ensuring that high availability is maintained in the resulting product. During the operation stage, online software upgrade is deemed necessary to reduce the overall system downtime by cutting the mean time to repair. The paper presents these three strands of investigations and highlights the results to date.

The included publication [P1], which is a refined version of [88], further develops the uniform framework of discussion by incorporating a view through the software product lines principle. The paper positions the carrier-grade platform in domain engineering and network element applications in application engineering. Focusing on the carrier-grade platform reference architecture, it advocates the use of standards-based availability management middleware for the high availability functionality building block. Through a gaps analysis, the paper identifies that three areas, namely high availability middleware capabilities, online software upgrade and dependability benchmarking, must be studied further in order to assess their impact on the development in terms of fault model, process, tool and programming environments. It also presents the application of Service Availability Forum’s Availability Management Framework and Software Management Framework to an example GGSN
(Gateway GGSN Support Node) network element for a GPRS (General Packet Radio Service) core network. The paper forms the backbone of the overall structure of this thesis.

The author is the sole contributor to this publication.

7.2 Fault Tolerant CORBA: A Telecommunications Perspective

The included publication [P2] gives the motivation of developing the then recently published Fault Tolerant CORBA specification by the Object Management Group, presents its overall architecture and features, and considers whether it is applicable from the telecommunications perspective. The paper elaborates on the typical real-time and high availability requirements of call processing functions in telecommunications systems. It provides example systems that deploy distributed object technologies such as CORBA in the telecommunications industry. The paper describes the basic concepts, fault tolerance properties, fault management, and logging and recovery management in a Fault Tolerant CORBA system. It also lists the known limitations of the specification. The paper concludes that although the specification appears to be satisfactory for use in telecommunications applications, there is a need for further work such as quantifying the performance of an implementation.

The author is the sole contributor to this publication.

Further investigation on using Fault Tolerant CORBA following this publication has concluded that a more focused service availability standard was needed in mobile network applications. This has led to a new standardisation effort as reported in the next included publication [P3].

7.3 On the Development of an Open Standard for Highly Available Telecommunication Infrastructure Systems

The included publication [P3] describes a three-tier organisation of end users, communications service providers and telecommunications equipment manufactures in the telecommunications market, and articulates the relationships among these groups through their corresponding expectations and requirements. The paper reports the formation of Service Availability Forum, presents the motivation and describes its focus. It defines the notion of service availability and shows the positioning of the planned middleware specifications with reference to the underlying platform and application above. The paper presents a proposal of an application interface that has the potential of meeting the Service Availability Forum’s mission. The presentation contains adequate details of the proposed features supporting three types of applications (non HA aware, HA aware, resource proxy), and a division of availability management and supplementary HA services at the middleware level. The paper concludes that Nokia is committed to open standards and describes its current active role in the Carrier Grade Linux Working Group.

The author is the sole contributor to this publication.
7.4 The Emerging SAF Software Management Framework

The included publication [P4] was the interim publication of the then upcoming Software Management Framework specification, aiming to reveal the baseline of the standard, and communicate its current status and direction. The paper presents the emerging Software Management Framework in the context of a Service Availability Forum ecosystem. It describes the Software Management information model in which an inventory of the available software versions, their types and installation locations are held. It also contains a bi-directional link to the Availability Management Framework’s Information Model to represent the current deployment configuration. The concepts of upgrade campaign, upgrade procedure and upgrade step are introduced, allowing for different upgrade methods to be specified. It discusses the software upgrade activity in terms of these concepts, together with the failure detection and handling support. The paper concludes with the general direction of the specification that includes different upgrade methods, treatment of compatibility issues, administrative and user upgrade APIs, and system-wide upgrade notifications.

The author’s contributions in this publication include the introduction, issues and challenges, related work, software upgrade activity and upgrade campaign.

Further development along this paper’s stated direction has resulted in the eventual publication of the Software Management Framework specification in 2007.

7.5 First Experience of Conformance Testing an Application Interface Specification Implementation

The included publication [P5] describes the first attempt of conducting conformance testing a Service Availability Forum’s Application Interface Specification implementation on a proprietary service platform for mobile communications applications. The paper reports an approach that is based on the IEEE Standard for measuring conformance to POSIX®. It gives an analysis of the requirements and guidelines of the standard, and explains the necessary adaptation for the Application Interface Specification. The paper describes the implementation under test and discusses the test method implementation. It uses the Application Interface Specification’s component registration function as an example, and shows the generated test instances and conformance log. The paper concludes that thorough level of testing seems to have the right balance of confidence and manageability of test cases.

The author’s contributions include the analysis and evaluation of the IEEE Test Method Specification, its adaptation to Application Interface Specification implementations, design of the conformance testing system and the sole writer of this publication.

The adapted conformance testing approach has since been adopted by the SAF Test open source project as reported by [86].
7.6 Robustness Testing Techniques for High Availability Middleware Solutions

The included publication [P6] presents automated robustness test suites generation for Application Interface Specification (AIS) based high availability middleware, describes a scenario-based robustness testing technique and evaluates the efficiency of the considered testing techniques. This paper introduces an AIS robustness testing framework in which different sources of inputs that can activate robustness faults of a HA middleware are identified, each of which is assigned a specific testing technique. It details the generic input testing, type-specific testing and scenario-based testing. The paper discusses the results obtained for the generic testing and type-specific testing over OpenAIS, an open source project for implementing AIS. It shows the relative complexity by comparing the number of lines in the source code of the robustness testing framework for the two testing techniques, and the effectiveness of each technique in terms of the faults detected per API function. The paper concludes that although simple testing techniques can detect robustness faults, there is a need for more complex methods.

The author has contributed to the overall direction of the investigation, the priority of exceptional input generator for the robustness testing framework, discussions on generic testing and type-specific testing.

7.7 Comparing Robustness of AIS-Based Middleware Implementations

The included publication [P7] uses the robustness testing framework developed in [P6] to conduct robustness testing on three implementations of the Availability Management Framework (AMF), OpenAIS-0.80.1, OpenAIS-trunk and Fujitsu-Siemens Computers’ SAFE4TRY. The paper describes the automatic template-based type-specific test generator for individual API functions, mutation-based sequential test generator for complex call sequences, and OS wrapper tool for intercepting and then injecting faults into system calls. It discusses results obtained from running the benchmark suites on the three AMF implementations, demonstrating and validating that robustness of AMF implementations can indeed be objectively compared. The paper also points out that by running the benchmark suite can uncover some other faults than robustness problems. The paper concludes that the novelty of this approach is the capability of the tools that generate test cases directly from the interface specification. The test suites, results and log have since been released to the open source community.

The author has contributed to the overall direction of the investigation, a tool concept of integrating the selection of a third party availability management software into the development cycle and verification of the test results.
The robustness test suites and the test results on the two versions of the OpenAIS implementations have since been contributed back to the project’s developer community.

7.8 Standards Contributions

As a technical representative of Nokia to the Service Availability Forum from 2001 until 2006, the author has contributed to the development of the Availability Management Framework and Software Management Framework specifications. These contributions form the basis of Chapters 4 and 5 in this thesis.

7.8.1 Availability Management Framework

The author has contributed to the first release of the Application Interface Specification [75], mainly in the Availability Management Framework through the joint submission team consisting of Sun Microsystems and IBM.

The author has contributed to all the discussions and took part in most of the decisions making process during the development of the joint submission, which was selected as the baseline for further development. He has also taken part and contributed to this subsequent development, which led to the publication of the first release of the specification.

The author has participated in the development of the AMF basic model of component, service unit, service group, service instance and component service instance; protection group; redundancy models and component capability models.

7.8.2 Software Management Framework

As the co-chair of the Software Management Specification Development group during 2004-2006, the author has contributed to establishing the requirements of online software upgrade and subsequently developing the Software Management Framework specification [78] with member companies consisting primarily of Ericsson, Fujitsu-Siemens Computers, Motorola and Clovis Solutions Inc.

The author has contributed to all the discussions during the specification development and took part in key decisions making. The specific contributions of the author were related to the upgrade campaign; upgrade campaign specification; finite state machines for upgrade step, upgrade procedure and upgrade campaign; failure detection and failure handling.
8 Conclusions

Now this is not the end. It is not even the beginning of the end. But it is, perhaps, the end of the beginning.

– Sir Winston Churchill

This chapter gives an analysis of the overall approach conducted in this research, assesses how well the research questions have been addressed, and speculates about future developments.

8.1 Analysis of the Overall Approach

The thesis has presented directions and the current status of developing standards-based service availability middleware for carrier-grade platforms that are deployed in the mobile networks industry.

The created constructs in this research are the Availability Management Framework, Software Management Framework and the associated robustness testing concepts and tools for supporting the selection of a COTS standards-based availability management middleware.

The main practical relevance of this research is reflected in the proposed solutions of availability management and online software upgrade have been standardised by the Service Availability Forum. In addition, the SCOPE Alliance [71], aims at accelerating the deployment of carrier-grade base platforms for service provider applications, has defined a baseline for a carrier-grade middleware profile [72][73] that uses the SAF’s middleware standards. Since the membership of the SCOPE Alliance consists of all the current network equipment providers, this has effectively endorsed AMF and SMF as the service availability standards in the mobile network industry. When all the technologies involved are matured and commercially viable to be deployed, a considerable and quantifiable return on investment is anticipated.

In addition, the developed availability management and online software upgrade capabilities for carrier-grade platforms have underlined the very essence of development for reuse in domain engineering of a software product line. The research results have shown that the service availability support in the platform exhibits the characteristics of variability, reuse, scalability, portability and extensibility that lead to development cycle time and ultimately costs reduction of products. The robustness testing concepts and tools have contributed to the understanding of the impact of adopting a standards-based service availability middleware on
the development life cycle of carrier-grade platforms. The research results have advanced the knowledge of incorporating robustness testing as a means of assessing COTS products that must be dependable, as well as projecting a path for creating a reusable dependability benchmarking suites as an asset that supports a software product line.

As a high-level, general validation of the overall approach followed by this thesis, an attempt has been made to ascertain whether the investigated topic areas merit any attention in the first place. A study conducted in [84] has argued that three industrial trends have impacted experimental research in dependability, addressed in the following:

The first observation is *shifting error sources*, which states that hardware failure rates are down, software failures have shot up, and planned outages have emerged as a prominent issue due to unavailability and disruption of services. The study of robustness testing and its potential to be extended to benchmarking the dependability of a software product lines, and online software upgrade support in carrier-grade platforms can be placed in this category.

The second observation is *explosive complexity*, which is primarily caused by the substantial growth in individual industrial segments, have resulted in a concept of synthesising ready-made components from various origins. The examination of using standards-based components in carrier-grade platforms for some control over the complexity issue belongs to this line of investigation.

The third observation is *global volume*, which necessitates dependability in systems management and includes the elimination of potentially error-prone manual operations. The development of a means of automating online software upgrade in this thesis falls into this group.

### 8.2 Thesis Questions Revisited

In spite of corroboration from other researchers in the field on the general level, it is necessary to evaluate specifically how well the thesis questions presented in Section 1.3 on page 6 have been addressed. In the following, each question will be examined again:

- **What kinds of service availability capabilities are needed in a carrier-grade platform for network infrastructure devices?**
  The capabilities must provide the applications with mechanisms for error detection, diagnoses and recovery, and the introduction of protection redundancy. In addition, online software upgrade with no, or minimal service interruption must be offered. Some kind of methods must be in place to manage and implement the redundancy model and upgrade strategy variations.

- **What types of constructs are worth standardising in a service availability middleware?**
  The types of constructs, together with the reasoning behind them, have been indentified as the service availability building blocks in Section 3.2 on page 26. The validation is in
what have already been included in the AMF and SMF specifications, which have gone
through the rigorous standardisation process. They are:

1. **Means of maintaining information about software entities and their relationships.** Information about software entities and their relationships are maintained through the AMF conceptual model of logical entities such as components, service units and service groups. The HA state per CSI shows the availability status, while the service group’s redundancy model reveals the kind of protective measures are in place. The entity types file as standardised in SMF is a descriptor for expressing versioning, compatibility and dependency information associated with online software upgrade.

2. **Means of configuring protective redundancy.** Configuring protective redundancy is carried out by AMF’s component capability model in which the decision of which redundancy model to use can be deferred to deployment time.

3. **Means of managing faults from their detection to recovery and repair.** Faults detection is provided by AMF’s programming API for error monitoring and reporting on components. Recovery actions are supported by AMF’s administrative API to restart or failover logical entities. Repair actions are performed by AMF’s administrative API to reload software or reboot a node.

4. **Means of replacing and upgrading software entities.** Upgrade campaign in SMF is used in the deployment phase to structure the upgrading of software entities from one configuration to another. The definition of failure detection and handling of the upgrade campaign execution ensures that service interruption is managed.

5. **Means of configuring upgrade strategies.** Configuring upgrade strategies is by means of selecting the appropriate upgrade scope, upgrade method and acceptable service outage through the upgrade campaign specification in SMF.

- *How can online software upgrade with minimum or no service interruption be supported on a carrier-grade platform?*

SMF cooperates with AMF which manages the availability of components while the system is being upgraded. Failure detection and handling during an upgrade operation are provided by SMF as protective measures. The level of service outage, both expected and acceptable, is visible and expressed through the upgrade campaign specification according to the required level of service availability.

- *What sorts of methodologies and tool concepts are needed to support a product selection process for a third-party, standards-based availability management software?*

Conformance testing and robustness testing have been identified as the required methodologies to be incorporated into the third-party COTS availability management
software assessment and selection process. Adding fault-load and workload pairs as input to the dependability benchmarking machine can broaden this process to cover individual products in a software product line. The robustness testing suites for the COTS availability management middleware thus become a reusable asset.

- **To what extent can a standards-based service availability middleware be utilised to construct a carrier-grade platform for network infrastructure devices?**

Technical evaluation on using a standardised service availability middleware to construct common platforms for network infrastructure devices during the rigorous SAF standardising process has confirmed that this approach is feasible. By means of the example GGSN case study, it has been illustrated that using AMF as the availability management middleware and SMF for the online software upgrade at the level of a common platform are applicable in practice. This additional validation has further concluded that this standardised service availability middleware approach can potentially be used in a software product line of network elements.

### 8.3 Future Work

In the short term, the following improvement of the three investigated strands presented in the thesis are anticipated.

AMF has become the dominant standard in the mobile network industry, with implementations from open source projects to commercially available products. Therefore, no significant future work other than maintenance of the published specification and the enhancement of existing products is expected. New standards development in this area are not needed due to the selection of the SAF specifications by the SCOPE Alliance for the mobile network equipment providers. Also the unfavourable economic outlook for the industry in the near term indicates that investment on standardisation similar to this scale is unlikely.

SMF is at the stage where feedback of using it in the field is required. Some forms of empirical studies are therefore urgently needed.

The support for COTS availability management product selection requires the creation of a representative, configurable workload and fault-load for the dependability benchmarking machine for each class of network element application. This allows for the tailoring of different measurement criteria for different products in a product line.

Given the breadth and complexity of the subject, the thesis has only focused on aspects of developing standards-based service availability middleware for carrier-grade platforms. There are plenty of opportunities for future research in the medium to long term. The following is a brief description of some of the identified ones.

- **Business analysis.** One business question has yet to be answered is how much a company saves by using standards-based carrier-grade platforms. The requires
quantifying the return of investment by collecting and analysing the costs on actual product development projects.

- **Tool support.** Support for designing and analysing network element applications that use AMF is anticipated. Some of the features may include the design trade-off analysis of how different service group redundancy models, service instance assignments and number of physical nodes may impact service availability. Similar support for applications that use SMF may offer the analysis of how resource requirements, structuring of an upgrade campaign, including its procedures and steps may impact the level of service outage.

- **Advanced upgrade.** The concepts of upgrade campaign, upgrade procedure and upgrade step are fundamentally sound and work well with conventional upgrade methods. However, there is not enough understanding of all the intricacies of how these conceptual entities interact with other entities on the availability management side. For example, how the combination of an upgrade scope with a service group redundancy model may impact the overall service availability? Is a formal relationship among all these conceptual entities needed? How can the inherent parallelism of multiple upgrade procedures be exploited? Are there any novel method through which more dependable and optimised ways of upgrading can be delivered? Further investigation may have the answers to some of these questions.

### 8.4 Concluding Remarks

Some of the results reported in this thesis may have the potential of being generalised and applied in other application domains. For example, Web services [91] as a platform have been increasingly supporting a wide range of demanding applications that require a high level of dependability. The traditional best-effort techniques in the Internet must be enhanced in order to deliver this expected quality. However, as Birman has pointed out there were still significant limitations in the Web Services standards dealing with reliability [12]. One viable source of input could be derived from the vast experience of dependability in the mobile networks industry.

Although Web Services have less stringent availability requirements [14] than those of the network elements in mobile networks, they both nevertheless share a number of similar properties. Therefore, AMF could be introduced as a standards-based support for service availability, conceivably brought in orthogonal to the current architecture without breaking it. Being typically third party software and subject to regular upgrades due to new features and/or bug fixing, Web Services can exploit SMF for upgrading Web Services with minimum, or no service interruption, and robustness testing and dependability benchmarking for the selection of commercially available Web Service platforms.
An architecture for managed upgrade has been advocated in [31] as a solution to address the issue of upgrading composite Web Services online while maintaining certain level of confidence. Some preliminary results of conducting robustness testing of Web Services have recently been reported in [90] and [49]. A more general but much longer term objective of assessing dependability has received much attention. A recent Framework Programme seven Coordination Action for Assessing, Measuring and Benchmarking Resilience (AMBER) [6] has undertaken to make dependability/resilience assessment a standard process. As an Advisory Board member, the author has provided input that is based on the robustness testing strand of this thesis to the AMBER consortium. A research roadmap and agenda representing the stakeholders’ views are expected to be made visible to the research sponsors, for example, in the European Commission’s ongoing process of defining priorities for its 7th Framework Programme.
References


Included Publications


Copyright ©2008 CRL Publishing. Reprinted, with permission from CRL Publishing Ltd.

Copyright ©2002 IEEE. Reprinted, with permission from IEEE.

Copyright ©2006 Springer. Reprinted, with permission from Springer.

Copyright ©2004 Springer. Reprinted, with permission from Springer.

Copyright ©2007 Springer. Reprinted, with permission from Springer.