Transparent and Adaptive Application Partitioning using Mobile Objects

A thesis submitted for the degree of

Doctor of Philosophy

Hendrik Gani B.App.Sc. (Hons.),

School of Computer Science and Information Technology,

Science, Engineering and Health College,

RMIT University,

Melbourne, Australia

14 April 2010

Declaration

I certify that except where due acknowledgement has been made, the work is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program; any editorial work, paid or unpaid, carried out by a third party is acknowledged; and, ethics procedures and guidelines have been followed.

Hendrik Gani

School of Computer Science and Information Technology

RMIT University

14 April 2010

Acknowledgements

First and foremost, I would like to express my gratitude to my senior supervisor Dr. Caspar Ryan for his unlimited support throughout my whole candidature. I am especially appreciative of his valuable advice and ideas which encompassed many aspects of my research, implementation and writing.

I want to thank my second supervisor Professor Zahir Tari, who has lent me support of various forms during difficult moments. Also thank you to my research consultant Dr Pablo Rossi, whose particular experience and knowledge in the area of adaptation and metrics is of significant benefit to my research.

Finally, I would like to thank all students who have contributed to the improvement of the MobJeX framework, which is fundamental to the development and realisation of my research work.

Credits

Portions of the material in this thesis have previously appeared in the following publications:

H. Gani, C. Ryan, and P. Rossi, "Runtime Metrics Collection for Middleware Supported Adaptation of Mobile Applications," in *Proceedings of 5th Workshop on Adaptive and Reflective Middleware, ACM Middleware 2006*, Melbourne, Australia, 2006, pp. 2-7.

H. Gani and C. Ryan, "Improving the Transparency of Proxy Injection in Java," in *Thirty-Second Australian Computer Science Conference (ACSC 2009)*, Wellington, New Zealand, 2009, pp. 43-52.

Contents

List of Figures

List of Tables

Abstract

The dynamic nature and heterogeneity of modern execution environments such as mobile, ubiquitous, and grid computing, present major challenges for the development and efficient execution of the applications targeted for these environments. In particular, applications tailored to run in a specific environment will show different and most likely sub-optimal behaviour when executed on a different and/or dynamic environment. Consequently, there has been growing interests in the area of *application adaptation* which aims to enable applications to cope with the varying execution environments.

Adaptive *application partitioning*, a specific form of non-functional adaptation involving distribution of *mobile objects* across multiple host machines, is of particular interest to this thesis due to the diversity of its uses. In this approach, certain runtime information (known as *context*) is used to allow an object-oriented application to adaptively (re)adjust the placement of its objects during its execution, for purposes such as improving application performance and reliability as well as balancing resource utilisation across machines. Promoting the adoption of such adaptation requires a process that requires minimal human involvement in both the execution and the development of the relevant application. These challenges establish the main goals and contributions of this work, which include:

1) Proposing an effective application partitioning solution via the adoption of a decentralised adaptation strategy known as local adaptation.
2) Enabling adaptive application partitioning which does not require human intervention, through automatic collection of required information/context.
3) Proposing a solution for transparently injecting the required adaptation functionality into regular object-oriented applications allowing significant reduction of the associated development cost/effort.

The proposed solutions have been implemented in a Java-based adaptation framework called MobJeX. This implementation, which was used as a test bed for the empirical experiments undertaken in this study, can be used to facilitate future research relevant to this particular study.

Chapter 1. Introduction and Rationale

The dynamic nature and heterogeneity of modern execution environments such as mobile, ubiquitous, and grid computing, present major challenges for the development and efficient execution of the applications targeted for such environments. More specifically, applications tailored to run in a specific environment will show different and most likely sub-optimal behaviour when deployed in a different context (e.g. a different device or network environment). Furthermore, given the dynamic nature of device usage, network connectivity, and power consumption, it is not practical to predict in advance the exact conditions in which an application may run. Consequently, there has been growing interests in the area of *application adaptation* [38, 87, 109] [144] [73] [186] [53], the aim of which is to enable applications to cope with varying execution environments by adjusting application/middleware behaviour according to the specific characteristics of the execution environment and the application itself. Such applications are commonly known as *adaptive* or *adaptable* applications.

At the minimum, the tasks required to perform adaptation can be categorised into: collecting contextual information (e.g. memory usage), making adaptation decisions based on collected information, and adapting according to the established decisions. Contextual information, which is also known as *context* [16] [50], refers to information about particular execution conditions which include characteristics about the running application (e.g. execution duration) and its execution environment (e.g. CPU usage). In order to remove the need for human intervention during application execution, it is important that *context* is collected automatically (e.g. by adaptive applications), and as such, one of the aims of this thesis is to address the automatic collection and management of context as discussed further in section 1.1.

A form of adaptation which is of particular interest to this thesis is *application partitioning*, which refers to the separation and distribution of relatively independent components (of an application) to multiple machines. Amongst the many benefits offered by such adaptation, which are discussed in detail in section 2.2, is the improvement of application performance through the distribution of application components to machines which match their specific system resource (e.g. processor, memory) requirements. The maximisation of the performance improvement gained from adaptive application partitioning (also known as the *effectiveness* of adaptation), is also a focus of this thesis and as such is discussed in section 1.1.

In addition to the aforementioned goals regarding the effectiveness and automation of adaptive application partitioning, the *overhead* involved in the development of applications supporting such adaptation is also of particular concern to this work. Consequently, this thesis proposes a solution for facilitating the development of adaptive applications through an

automatic process involving injection of adaptation capabilities into existing applications as discussed in more detail in section 1.1.

1.1 Scope and Goals

The scope and primary directions of this research are discussed in this section. Throughout the discussion, a brief description of the major issues and limitations in the current state of related research is provided in order to outline specific high-level goals, which serve to establish the research questions presented in section 1.2. On the other hand, a more detailed discussion of related work, which serves to provide a comprehensive analysis and comparison between existing approaches, is provided in the literature review presented in Chapter 2.

As mentioned, this thesis focuses on a specific form of adaptation which is achieved via application partitioning. A popular approach for partitioning an application is through *object mobility* (also known as *object migration*) [134] [79] [147], which refers to the ability of an object-oriented application to migrate some of its constituent objects (known as *mobile objects*) to different machines. The mechanics concerning object migration have been sufficiently investigated in previous work (e.g. [134], [79], [147]), thereby allowing this thesis to instead focus on issues related to decision making (e.g. determining objects to be migrated) and context management (e.g. collecting context required for decision making).

The manner in which adaptation decisions are made depends heavily on whether the adopted solution applies a local or global adaptation scheme. In *local adaptation*, which is the focus of this thesis, each collaborating machine shares adaptation responsibility by making decisions regarding the (re-)location of objects presently residing on the machine. This is in contrast to the more traditional *global adaptation* strategy (e.g. [73] [186] [53]), wherein adaptation decisions are made by a single centralised component, which provides a performance bottleneck as well as a single point of failure.

Since, in the case of local adaptation, (partial) decision making is carried out independently by each participating machine, producing good quality decisions is a major challenge. Note that the quality of adaptation decisions is often measured in terms of the *effectiveness* of achieving a specific goal(s), which in the case of this work is to improve application performance. Consequently, improving the effectiveness of application partitioning in a local-adaptation scheme is one of the research questions in this thesis as discussed further in section 1.2.

Additionally, due to the distributed nature of local adaptation, there exist complexities related to the management of context, in terms of when, where and how context (required for adaptation) is collected, delivered, and/or exchanged. Furthermore, there is also the challenge for automating the whole process involved in the management of context, for the purpose of

supporting *continuous (online)* adaptation, which refers to the case in which an application continuously observes and adapts to changes in its execution context. Such adaptation is preferred over the *offline* (i.e. prior to application execution) and the *startup-only* approach (i.e. only once during application start-up), because it is more suitable for addressing unpredictable (e.g. dynamic) execution conditions (which are commonly exhibited by modern computing paradigms such as mobile computing), as discussed in section 2.1. Consequently, this work aims to address the aforementioned challenges through the investigation of relevant issues (e.g. how context should be represented) as well as the formulation of an efficient solution (e.g. for collecting context), as outlined in section 1.2.

In this work, context is collected in the form of *metrics*, the purpose of which is to provide quantitative measures of certain attributes of the execution (e.g. CPU usage, execution duration), as discussed further in 2.4. An advantage of such an approach is that it allows the obtained measures to be used in mathematical formulas for the purpose of making adaptation decisions, as demonstrated in existing work on local-adaptation [144], which is used as a baseline for the (improved) decision making algorithm proposed in this thesis.

The formulation of generic solutions for adaptation decision making and metrics management that are applicable to various object-oriented applications, is made possible because application partitioning is a form of *non-functional adaptation*, which as explained in section 2.1, refers to the alteration of behaviour that is not directly related to the domain-specific functionality or services provided by the application. One benefit of this type of adaptation, in comparison to *functional adaptation* (e.g. auto-adjustment of video playback quality), is that its realisation does not rely heavily on the presence of domain-specific implementation/knowledge.

As such, not only does this promote the reusability of the formulated solutions, but it also opens the possibility of reducing software development effort through the provision of generic support from development frameworks and middleware. Such an advantage is highly relevant to this work, since the concerned adaptation approach (application partitioning) requires additional functionality (e.g. metrics collection, remote communication) to be implemented in the adapted application (as discussed in detail in section 2.1), thereby implying that additional development effort is required despite the endeavour to implement common functionality in external components, e.g. middleware.

Various frameworks (e.g. [53] [134] [51] [147]) have been proposed in previous work to facilitate the development of applications supporting dynamic (i.e. either adaptive or requiring human decisions) application partitioning via object mobility. Different frameworks differ in terms of the supported functionality (e.g. adaptation, context collection, concurrency management) as well as the provided development transparency (i.e. how much developer involvement is required), as reviewed in detail in section 2.5. A popular approach adopted by

existing frameworks to address development transparency is through *code transformation*, which in the context of this discussion, refers to the automatic process of transforming (the source/byte code of) a regular non-adaptive application into an adaptive application.

A similar approach is adopted in this work, wherein code transformation is used to facilitate *functionality* or *capability injection*, a process referring to the automatic injection of adaptation functionality (e.g. metrics collection) into a regular (object-oriented) application. In this approach, the majority of injected functionality is encapsulated in object *proxies* (i.e. intermediary objects which bridge the interaction between two objects), which as discussed in detail in section 2.5.1, play important roles in adaptive application partitioning. Despite also addressing other aspects, such as flexibility and portability, the proposed solution is primarily concerned with the transparency of capability injection in terms of not breaking existing code (i.e. the original application) in an effort to minimise development effort, as emphasised in section 1.2.

In order to further promote development transparency, Java, which is a cross-platform technology (thus providing better support for applications running on heterogeneous platforms as discussed further in section 2.5.2), is the main focus of this thesis to the extent that the proposed solutions: 1) were implemented and evaluated using Java, and 2) are aimed at facilitating adaptive partitioning of Java applications.

1.2 Research Questions

Having discussed the background, scope and goals of this research, the following are the three distinct research questions that this thesis aims to answer:

1. How can the effectiveness of application partitioning in a local-adaptation scheme be improved?

   In order to answer this question, an existing local-adaptation algorithm originally proposed by Rossi and Ryan [144] was investigated. A number of limitations in the original algorithm were identified and addressed, which consequently results in the formulation of the proposed algorithm discussed in Chapter 3. Changes in the proposed algorithm include those for facilitating: 1) the use of alternative metrics which more accurately reflect the represented attributes (e.g. the CPU requirement of an object), 2) the introduction of new metrics to address specific limitations of the original algorithm (e.g. unnecessary object migrations), and 3) the incorporation of additional information into adopted metrics (e.g. the degree of interaction between individual objects) in order to enable more complete understanding of the present execution conditions.

These improvements result in more effective adaptation in terms of the improved performance (i.e. reduced execution duration) of the adapted application, as confirmed in the evaluation in Chapter 5, which demonstrates the effectiveness of the proposed algorithm in scenarios involving heterogeneous and dynamic execution conditions.

2. How can metrics be collected and managed automatically in order to support the live execution of adaptive application partitioning?

   Answering this question, which is the focus of Chapter 4, involves addressing issues related to: 1) the collection of metrics required by the extended algorithm and 2) the delivery of collected metrics to the distributed adaptation engines. The proposed metrics management solution, which does not require human intervention (during application execution), is particularly concerned with the efficiency of the involved tasks as well as the accuracy of the collected metrics, because an efficient solution minimises its impact on the performance of the running application, whereas accurate metrics enable more optimal decision making (thus more effective adaptation). This work further extends the notion of metrics accuracy through the consideration of the temporal characteristics (e.g. recentness) of metrics, which also serve to improve the quality of decision making.

3. How can the transparency of the injection of adaptive application partitioning functionality be improved?

   This question is answered through the provision of a solution for automatically injecting object proxies into regular object-oriented applications, the core contribution of which is concerned with the transparency of object proxies in terms of allowing proxies to be injected without breaking existing application code, as presented in Chapter 6. While it is inevitable that the injected proxies (and the included adaptation functionality) alter the original functional behaviour of the application, the solution pays particular attention to the prevention of undesirable changes of non-application-specific semantics, such as the polymorphic behaviour of methods/invocation.

   The second part of the injection solution, which is addressed in Chapter 7, concerns the code transformation tasks (e.g. modifying existing code, generating new code artefacts) that are required by the proposed proxy transparency solution. The code transformation solution addresses requirements specific to the development of adaptive applications, which often involves manual customisation (e.g. extension, fine-tuning) of the injected/produced adaptation functionality. Additionally, issues related to the transformation of distributed applications for execution in heterogeneous

execution environments, which are inherent in adaptive application partitioning, are also addressed.

1.3 Contributions

As implied from the discussion in sections 1.1 and 1.2, this study serves as an exploration and investigation into different but related research areas, which include adaptation, application partitioning, and software metrics, for the purpose of addressing the heterogeneous and dynamic nature of the execution environments of contemporary applications. A primary outcome of this study is the design and implementation (in a supporting middleware and framework) of a completely automated solution for enabling runtime/online adaptation via application partitioning, which (as discussed in section 5.3) is empirically shown to improve application performance in unpredictable and dynamic execution environments. The solution does not require domain-specific knowledge and therefore is applicable to various applications, including those which exhibit dynamic behaviour (i.e. change its behaviour during execution). Additionally, despite primarily targeting specific local-adaptation algorithms, the core of the proposed metrics management solution is generic and is therefore applicable to other adaptation algorithms and strategies (e.g. global adaptation), as discussed further in section 4.1.

A further contribution of this study is the investigation of issues related to transparent capability injection, the outcome of which allows adaptation capabilities to be injected into existing applications with significantly reduced human involvement. Such a contribution plays an important role in the adoption of the proposed adaptation solution, since this allows the solution to be used with minimal effort, which not only implies lower software development cost but also indicates better software quality (e.g. fewer human errors). The injection solution is generic and as such can be applied to other forms of adaptation, such as that achieved by dynamically swapping a particular object with another compatible object.

The proposed solutions (i.e. adaptation, metrics management, and capability injection) as well as the complementary functionality (e.g. error handling) have been implemented in a Java-based mobile object framework called MobJeX. Not only did this implementation enable the execution of the empirical experimentation presented in this thesis, it also serves as a basis for future studies on related topics. Furthermore, although not yet of production standard, the implemented solutions are sufficiently functional that various real-world applications may benefit from the provided adaptation support.

1.4 Thesis Structure

This thesis is structured as follows. Chapter 2 reviews existing literature on related topics, which include application adaptation, application partitioning, object mobility, context management, software metrics, and capability injection. The literature review serves to outline specific issues and limitations in previous work (e.g. using inaccurate metrics, involving substantial development effort) as well as to provide a background on general approaches of relevance to this work (e.g. local versus global adaptation, source-code versus byte-code transformation). Chapter 3 involves a detailed discussion on the adopted local-adaptation solution with particular emphasis on the differences between the proposed decision making algorithm and the existing algorithm proposed by Rossi and Ryan [144]. Chapter 4 presents a solution for supporting the collection and management of specific metrics required by the adaptation algorithm proposed in chapter 3. Chapter 5 presents an evaluation of the effectiveness and overheads of the adaptation and metrics management solutions proposed in chapters 3 and 4. Chapter 6 addresses various issues related to the transparency of proxy injection (such as the structural compatibility between a proxy class and the proxied class), whereas chapter 7 discusses the code transformation solution for facilitating such injection. Finally, chapter 8 concludes this thesis with a summary of contributions and a discussion of future work.

Despite primarily focusing on Java (as mentioned in section 1.1), as much as possible, the discussions in subsequent chapters are presented in a general rather than Java-specific context. Consequently, non-Java related work is also included in the literature review in Chapter 2 to establish the background and scope of this research. On the other hand, when analysing existing approaches, the survey primarily focuses on Java-related work due to its direct relevance to this thesis. The adaptation and metrics management solutions proposed in Chapter 3 and Chapter 4 are generic although complementary discussions on the implementation aspects are specific to Java. The experiments discussed in Chapter 5 were undertaken using Java, but the majority of the resulting conclusions should still apply to the case where other technologies were used instead (e.g. the proposed algorithm is more effective in certain execution environments). On the other hand, the capability injection solutions presented in Chapter 6 and Chapter 7 are specific to Java and therefore can only be discussed in the context of Java.

Chapter 2. Literature Review

This chapter reviews existing work in the areas of *application adaptation* (e.g. [87], [38], [109]) as well as *context awareness* (e.g. [188], [41]), which show significant similarities, although the two areas focus on different issues. Application adaptation is mostly used to address variability in hardware resources such as processor load, whereas context-awareness primarily concerns information about the user of the application as well as his/her surroundings, although the term *context* also encompasses information used in application adaptation (e.g. hardware resources) as discussed in detail in section 2.4.

This review also includes work on specific issues related to the automatic management of context (required by adaptation) and the development of applications supporting adaptive application partitioning. It serves to compare and analyse existing approaches for the purpose of: 1) emphasising quality attributes (e.g. portability, development flexibility) that are of interest to this work, and 2) establishing a basis for the adaptation solution proposed in this thesis.

To begin, section 2.1 provides a high-level discussion of different types of adaptation and the different ways in which an application can adapt. Next, section 2.2 discusses *object mobility*, which is of particular interest to this thesis due to its role in supporting adaptive application partitioning. Sections 2.3 and 2.4 investigate alternatives for performing adaptation and managing the required context. Lastly, sections 2.5 and 2.6 are concerned with the minimisation of the effort/cost of developing applications supporting adaptation via object mobility.

2.1 Adaptation Variations

Due to the wide-ranging benefits and promises offered by application adaptation (e.g. automatic load balancing [15] [125], adaptive application partitioning [87] [144]), existing work in the area of application adaptation varies greatly in terms of approach and complexity, depending upon the aims of a particular study. For instance, specific adaptation tasks such as collecting contextual information (also known as *context*), making adaptation decisions and performing the actual adaptation, can be executed in different software components depending on where they are implemented [150].

According to the classification of [150], at one extreme is *laissez-faire* adaptation, which involves adaptation tasks performed entirely by the application itself. At the other extreme is *application-transparent* adaptation (e.g. [70] [143]), in which adaptation is carried out independently of the application, by separate entities such as libraries, middleware systems or the operating system. Finally, adaptation that falls somewhere between these two extremes is

termed *application-aware* adaptation (e.g. [87], [38], [186]), in which adaptation is performed collaboratively between the application and external support components.

Laissez-faire is the least desirable approach because it increases the complexity of the application, thereby potentially affecting both performance and maintainability. Consequently, the application of such an approach was not found in the literature. On the other hand, although *application-transparent* adaptation reduces developer effort and does not require changes to the original application, the range of adaptation capabilities that can be supported is limited since the adaptation does not consider the characteristics of a specific application. Consequently, such an approach is not suitable for adaptation achieved via object mobility [87], which requires application-specific information such as runtime coupling[1] between objects [9].

Application-aware adaptation, which is a compromise between the two extremes, addresses this limitation by delegating common functionality such as thread management and remote communication, to external components, while leaving application-specific operations to the application itself. Such an approach supports different forms of adaptation with varying degrees of complexity (as discussed later in this section), thereby introducing new challenges related to the development of adaptive applications, the complexity of which positively correlates to that of the supported adaptation. Consequently, one aspect that is of particular interest to this thesis is *development transparency*, which refers to the amount of developer effort required to develop adaptive applications, as discussed further in section 2.6.

Adaptation can be classified into *offline* and *online* depending on the phase in which it is performed. *Offline adaptation* (e.g. [186] [84] [167] [110]) is performed prior to the execution of the application, i.e. during application deployment, whereas *online adaptation* (e.g. [87], [109], [189], [65]) is carried out at run time. Offline adaptation is achieved by tailoring the original application code to realise the targeted runtime behaviour, which does not necessarily change during application execution (examples are provided in the next paragraph). This type of adaptation is *application-transparent* since adaptation is never performed by the application but rather by an external entity (e.g. a compiler or framework).

In its simplest form, offline adaptation can be used to introduce new behaviour to specific applications as applied in [167], which addresses the adaptation of a single-user application into a collaborative multi-user application. Other applications of offline adaptation include modifying functionality/services provided by the application as well as managing the distribution of application components to match the static characteristics of the target device(s).

[1] Object coupling may also be referred to as object dependency [40] J.-L. Chen, F.-J. Wang, and Y.-L. Chen, "Program Slicing: An Application of Object-oriented Program Dependency Graphs," in *Proceedings of the Technology of Object-Oriented Languages and Systems*, Beijing, China, 1997, p. 121.

For example, in the absence of adequate support on the target device, video rendering functionality can be dropped from a multimedia player, thus leaving it with only audio playback functionality [84].

Furthermore, application code can be modified in such a way that certain functionality, encapsulated as runtime components (i.e. application objects), is distributed to different devices in an attempt to match resource requirements (e.g. processor usage) to device characteristics (e.g. processor capacity) as addressed in [84] and [186]. Note that the latter work [186], which investigates distribution/migration of live application objects, in fact uses a combination of offline and online approaches. In this case, adaptation decisions are made at deployment-time (i.e. offline) using information acquired via code analysis and offline profiling, whereas object migration is performed during application startup. The migration operation can be classified as *startup-only* adaptation as opposed to the more conventional *continuous* adaptation.

Both *offline* and *startup-only* (online) adaptation approaches are suitable for addressing the issue of platform heterogeneity (e.g. various device capabilities) rather than the issue of dynamic execution context (e.g. changing resource usage/availability), which is the primary aim of *continuous* (online) adaptation. Furthermore, in continuous adaptation, the application also adapts (albeit indirectly) to the heterogeneity of the execution environment, although adaptation might not occur immediately depending on the specific behaviour of the adaptation algorithm and the availability of the required information/context. Consequently, since modern computing paradigms such as pervasive, mobile, and grid computing, involve execution environments that are heterogeneous, unpredictable, and dynamic, continuous (online) adaptation is the focus of this thesis and thus of subsequent discussions.

There are different ways in which adaptation can be performed, which include modification of certain behavioural parameters, composition of application components, and mobility/migration of application components. Adaptation via parameter manipulation was applied in [116] and [24] to adjust the resource consumption (e.g. network usage) of the application based on the availability of the relevant resources. Such behaviour is appropriate for QoS (Quality of Service) applications such as adaptive image servers [24] or video playback applications [120] [21], wherein service quality can be sacrificed for reduced resource consumption. On the other hand, there has been increasing interest in more versatile adaptation approaches such as those achieved via component *composition* and *mobility*. Adaptation via component composition involves substituting specific application components (e.g. objects, services) with compatible replacements as addressed in [36], [78], [118], etc. On the other hand, adaptation via component mobility is achieved via physical distribution of certain application components (e.g. objects, executions/threads) such as that applied in [87], [186], and [148].

Component composition is often used for adaptation concerning the modification of specific functionality of a given application, which is hereinafter referred to as *functional adaptation* (e.g. [109], [35], [159]). On the other hand, component mobility is generally used in *non-functional* adaptation, which refers to the modification of operations that are not directly related to the functionality/services provided by the application. One important difference between the two types of adaptation is that the realisation of functional adaptation relies heavily on the presence of domain-specific implementation, whereas this is not necessarily the case in non-functional adaptation.

As such, it is difficult (if at all possible) to implement functional adaptation that works well with different types of applications, despite efforts made to generalise the particular functionality. Examples of such cases can be found in work related to *content adaptation* [35] [116] and *protocol adaptation* [109] [158] [11]. Content adaptation refers to the dynamic transformation of certain display content of the application (e.g. images, text) according to the characteristics of the host device. In contrast, protocol adaptation refers to the adjustment of the protocol that is used for communication between multiple devices (e.g. clients and servers), which generally requires prior negotiation between the devices.

As an example of *content adaptation*, MobCon [35] implements a solution for adapting a server-based image from its original form, to a resolution and colour depth matching the characteristics/capabilities of a specific client device. Although this approach can be applied to arbitrary digital images as opposed to specific image files, the solution does not necessarily apply to applications displaying different types of media such as video. Similarly, Fractal [109], which addresses *protocol adaptation*, considers gzip compression as a possible approach/implementation for transferring data between a server and a client device. Compared to sending raw data (i.e. without intermediary processing), gzip reduces the size of transferred data at the expense of higher computing overhead on both the sending and receiving devices. Again, although the gzip protocol applies to any kind of data, for certain data types it is not as effective as specialised compression algorithms, for example the mp3 format which specifically targets music files.

To summarise, due to the domain-specific nature of functional adaptation, application-specific adaptation functionality needs to be provided for different types of applications. Consequently, this increases developer effort even in the presence of supporting tools/components such as middleware platforms and frameworks. *Adaptation middleware* such as MADAM [55], MobiPADS [36], and MIDAS [135], serve to facilitate the execution of adaptive applications by managing common operations such as component/service recomposition [55] [36]. Although this alleviates developers from the intricacies of implementing common runtime operations, support for the development of application-specific adapta-

tion functionality (e.g. different compression algorithms as previously explained) is generally lacking.

On the other hand, *adaptation frameworks* such as Fractal [109], CASA [118], MoCA [145], and others [133] [157] [117] [155] [37], simplify the development of adaptive applications by providing support via libraries/APIs and/or automatic code transformation. The provided support is generic and therefore significant involvement from the application developer is often still required (to provide domain-specific adaptation functionality). Note that the distinction between standalone middleware and application development frameworks has become less clear because many recent middleware platforms perform byte-code transformation (to simplify application development) and similarly, most frameworks also include one or more middleware components. Consequently, subsequent discussions use the terms middleware and framework interchangeably unless explicitly specified otherwise.

As previously mentioned, in contrast to functional adaptation, non-functional adaptation is not as heavily dependent on domain-specific implementation since it targets operations that are not directly related to the domain functionality/services provided by the application. Operations of this type include those concerning application components such as mobility [87] [186] [148], [191] [147] [80] [134], self-healing [111] [23], and replication [165] [111], as well as those related to the underlying middleware such as concurrency management and service discovery (e.g. [70] [143]).

Application component mobility/migration, which is widely known as *code mobility*, is an area that is of particular interest to this thesis because of the many potential benefits that it offers, which as reviewed in [58] and [191], include load balancing and improving application performance, robustness, availability, scalability, etc. Furthermore, as shown in existing work [87] [186], application-specific implementation, which is a requirement (and thus a limitation) in functional adaptation, is not necessary in such adaptation, although the presence of domain-specific knowledge could be beneficial (e.g. for fine-tuning). Note that this thesis focuses on adaptation carried out via mobility/migration in response to changes in the original execution environment as opposed to *adaptive migration* [44], which refers to adaptation executed in response to the changes caused by the migration of an application or its components.

Code mobility can be applied to non-executing (i.e. static/stateless) code such as Java classes or applets [169], as shown in [38], which presents a solution for offloading class compilation to a remote machine in an attempt to reduce the energy consumption of the original device (i.e. an energy-constrained machine). Other work of this type includes mobility concerning stateless services [128] [127] [114] [142] which is performed at the level of Java classes or OSGi modules [124]. Such adaptation is characterised by the relatively simple tasks that are required, such as collection and analysis of attributes at class level as opposed

to at the finer-grained object level. The application of stateless code migration for facilitating adaptation is limited and rare, and consequently subsequent reviews concern mobility involving stateful or executing (i.e. live) application components such as threads or objects, which is the focus of this thesis.

Stateful code mobility can be categorised into *strong* and *weak* mobility [58]. Strong mobility refers to the migration of the current execution state of an application as well as the migration of the code and data required to continue the execution after migration (i.e. in the destination machine). In contrast, weak mobility does not include the migration of application execution state, thereby imposing constraints, such as requiring execution to be restarted in the destination machine, or prohibiting components from executing during migration. Conceptually, strong mobility often involves suspending and resuming the currently executing thread and thus is synonymous with the terms *execution migration* or *thread migration* [139] [192] [56]. Note that strong mobility can be extended to migrate all threads of the application, thereby achieving *full mobility* [20], which is synonymous to *process migration* [115] [66] [161] [33] [81] or *application migration* [190] (a term commonly used in Cloud Computing [184]).

In practice, due to the lack of native support provided by existing platforms (e.g. compilers, virtual machines), different techniques (e.g. [139], [122]) with varying degrees of complexity, portability, and efficiency, have been used to realise *strong mobility* and *full mobility*. For example, solutions used in Amoeba [161] and MPVM [33] [32] require modifications to the underlying operating system and thus are only applicable to homogeneous operating systems. On the other hand, specialised virtual machines were presented in [192] (i.e. JESSICA2) and [139] (i.e. Mobile JikesRVM), which although can be used independently of the underlying operating systems, are still limited in terms of portability due to the lack of ubiquity compared to standard virtual machines, such as those implemented according to Java or .NET specifications.

In contrast, [59] applies a portable solution that enables migrations at particular execution points (also known as checkpoints) specified by the application developer, which as a consequence reduces *development transparency*. A more transparent solution relying on automatic insertion of checkpoints is used in [179] and [122], at the expense of longer execution time due to the lack of domain-specific knowledge. On the other hand, flexibility was sacrificed for simplicity and performance in [57] by inserting checkpoints only at the entry point of thread executions, i.e. in `java.lang.Thread.run()` for the case of Java applications.

On the other hand, weak mobility involves less complexity since it does not require the migration of low-level execution state. Weak mobility is often used in object-oriented applications, wherein application objects (e.g. Java objects) are generally viewed as the smallest migratable units. Nevertheless, migration at a finer granularity (i.e. method level) is feasible

as investigated in [67], however the solution is somewhat limited since it does not address the automation of the adaptive decision making process, and lacks portability due to its dependence on a modified JVM. This thesis focuses on *object migration* (e.g. [87], [186]) because it is believed to be applicable to a wider range of application domains, since objects in a properly designed/implemented object-oriented application represent a cohesive application component encapsulating specific execution logic (i.e. methods), as well as the required data (i.e. fields[2]). Existing work on object migration is reviewed in section 2.2 with emphasis on specific issues related to application adaptation.

Note that even though objects do not define active executions (which are represented by threads instead), object migration might be used in the underlying implementation of strong mobility as shown in [20]. Consequently, even though the contribution of this thesis was primarily developed around the concept of weak mobility, the presented solution and practical application could be extended to support strong migration.

2.2 Adaptation via Object Mobility

Object mobility has been used in various application domains such as mobile agents [174] [74] [69] and application partitioning [108] [186] [51]. Object mobility facilitates the distribution of mobile agents for enabling autonomous execution of tasks such as information retrieval [69], on target machines. Adapting the distribution of mobile agents often involves specialised solutions as shown in [174], which focuses on moderating the population of agents in order to control resource consumption.

On the other hand, application partitioning, which involves distributing constituent objects across multiple host machines, is more generic since it applies to most object-oriented applications. In this approach, the distribution of an object is based on specific criteria, such as matching the resource requirements of an object to the resource availability of a machine. As argued in [108], application partitioning provides a more portable alternative for component/object distribution in comparison to solutions requiring changes to the underlying platforms, which include distributed-shared-memory (DSM) systems, such as cJVM [8], Jackal [181], and Emerald [94].

Application partitioning can be used in *non-adaptive* or *adaptive* schemes. Non-adaptive partitioning is applied in [108], in which application objects are manually distributed (e.g. by a system administrator) based on their roles. For example, objects accessing peripherals, such as projectors or interactive whiteboards, are migrated to machines in the vicinity for faster responses. Other examples include [131] and [132], which propose solutions for automati-

[2] In object-oriented programming, *fields* are also commonly known as *member variables* or *data members.*

cally partitioning a sequential application into a distributed/asynchronous version for the purpose of promoting parallel executions on multiple machines. However, even though the partitioning was automated, the support for automatic distribution of the resulting partitions (i.e. groups of objects) was not present. Note that this thesis refers to *sequential applications* as those that run one main thread but may also have supporting threads, hence are not necessarily single-threaded.

On the other hand, *adaptive* application partitioning can be categorised into *offline* and *online partitioning* depending on the type of information (i.e. static versus dynamic) used for making partitioning decisions. For instance, [186], [160] and [46] present solutions for partitioning applications based on static device information (e.g. memory size) as well as approximated application characteristics (e.g. memory consumption of objects/classes, interaction/dependency between objects/classes), which is acquired via analysis of application code. Although offline partitioning solutions provide meaningful initial placement of objects, they do not cope well with changing/dynamic user or application behaviour and resource availability.

An alternative approach is online application partitioning, which uses runtime information or context, allows an application to adaptively (re)adjust the placement of its objects, i.e. according to changes in execution context. For instance, [180] and [14] apply a solution for improving the reliability of partitioned applications by automatically retracting distributed objects in order to bring a partitioned application to its original state upon context changes, such as disconnection of participating devices. Despite addressing automatic object retraction, which is achieved by reversing the process of object distribution, the automation of initial object distribution was not discussed.

[148] presents a performance-oriented (i.e. response time) object-distribution solution, which considers runtime processing information such as CPU load and object execution intensity. However, the application of the algorithm is limited in practice since it does not consider network conditions and object interaction, which are major factors in the performance of a distributed application. As such, this solution is a relatively simple form of application partitioning, since without the analysis of object interaction, objects are distributed independently of each other, thus eliminating the notion of partitions.

In contrast, although [73] considers object interaction for minimising remote communication overhead, it does not consider processing (i.e. CPU-related) factors since its main objective is to relieve the memory constraint of the mobile device. Moreover, the presented solution targets a specific partitioning scenario, in which only two devices are involved: a memory-constrained device and a more powerful nearby machine. In addition, the solution requires modified JVMs which limit its applicability in heterogeneous environments since different devices might run different JVM implementations and versions.

Although work on object replication such as [189] and [165], shares similarities to application partitioning since replication in distributed systems requires object mobility, such work mainly focuses on aspects such as synchronisation between replicas and the intensity of read/write operations, which are not relevant to this thesis. Object mobility was also used to facilitate parallel executions, as demonstrated in [63], in which *active objects* (i.e. objects that execute asynchronously/independently of other objects) are distributed to different machines. The distribution is done adaptively to match the resource consumption of a particular active object to the resource availability of a given machine. Such an approach not only promotes parallel execution, but also ensures that objects execute on the most suitable environment/machine. However, this burdens the application developer since applications have to be tailored to fit into the active object programming model.

In comparison, [87] and [52] provide better development transparency since their work automatically analyses and groups objects based on their relevance to certain application threads (also referred to as tasks). Next, each task/thread and the related objects (i.e. those heavily used by the thread) are assigned to a specific machine in order to parallelise execution of independent threads. Such adaptation is however domain specific and thus benefits only a limited set of applications, i.e. those that can be partitioned into smaller computation units that execute concurrently.

On a related note, adaptive load balancing solutions [51], [57] [53] [126] should be applicable to sequential applications despite parallel applications being the targeted domain (e.g. [51], [57], and [53]), since no thread-specific information was used in the decision making. In [51], objects are offloaded to other machines when the processor or memory load of the source machine hits a threshold specified by the application deployer. However, as is also the case with [57] and [126], specifics of the actual load balancing algorithm were not provided, thus it is not known whether the algorithm simply picks random objects for offloading or considers the processor/memory consumption of individual objects, which introduces more complexity in the decision making and the collection of context information as addressed in Chapter 3 and Chapter 4.

On the other hand, although processor consumption of objects was considered in [53], it was only estimated using invocation frequency (i.e. the frequency at which the methods of an object get invoked), thereby loosely reflecting processor consumption due to variations in object/method implementation and input parameters. Another limitation of [53] is that a homogeneous network is assumed wherein all machines have the same resource (i.e. processor) capacity. Finally, network load and object interaction, which are major contributors to the overhead of distributed applications, were not considered in either [51] and [53].

Many of these limitations were addressed in [144], which proposes a multi-purpose algorithm, which at present, supports adaptation for improving application performance (i.e. re-

sponse time) and for balancing resource utilisation (i.e. processor, network, and memory). Consequently, the algorithm considers context related to processor, network, and memory utilisation as well as context related to the running application such as object/method execution and interaction. This context is represented as *metrics*, which serve to quantify execution attributes such as processor load or object interaction intensity, as discussed further in section 2.4.

In such adaptation, metric accuracy, which refers to how closely a metric reflects an attribute/condition, is important, because migration/distribution will result in improved performance (i.e. shorter response time) only when object characteristics (i.e. resource consumption) are properly matched to machine characteristics (i.e. resource availability). In contrast, the distribution of parallel applications which exploit the availability/existence of multiple processors, will likely result in better performance even in a suboptimal object distribution scenario. Consequently, a solution which addresses the efficiency and accuracy of context/metrics collection is proposed in Chapter 4.

Unlike most existing approaches, [144] uses the *local adaptation* (i.e. decentralised) approach, which as will be discussed further in section 2.3, provides better reliability and scalability than the traditional *global adaptation* (i.e. centralised) approach. Consequently, this thesis uses the solution proposed in [144], as a basis for the development of ideas/concepts for achieving adaptive application partitioning via object mobility. Although the evaluation of the base/original local adaptation solution [144] shows promising preliminary results, the solution has several limitations that are of particular concern to this thesis, which include the absence of a solution for automating adaptation (e.g. automatic collection of context) and the use of impractical/inaccurate context (which results in sub-optimal decision making). The high-level adaptation strategy adopted in [144] is described in 2.3, whereas the operational specifics of the algorithm are discussed in section 3.1.

2.3 Adaptation Strategy

Since the adaptation solution proposed in this thesis is derived from the solution presented in [144], for clarity and brevity, the base solution will hereinafter be referred to as the *original solution*, whereas the extended version which is presented as a contribution of this thesis will be referred to as the *proposed solution*.

Adaptation can be performed *reactively* or *proactively*. The *original solution* [144] uses a *reactive* approach, which refers to the execution of adaptation after the triggering event (i.e. changes of execution context). This is currently the more widely used approach in comparison to the *proactive* approach, in which an application adapts based on predicted future events.

Nevertheless, the proactive approach serves an important role for dealing with execution failures, because it allows applications to anticipate and address future failures. An example of such adaptation was investigated in [78], which focused on service-based applications wherein required services are sometimes dynamically discovered at runtime. A failure/deviation in terms of functionality or quality of service could occur when newly discovered services do not behave as expected. Consequently, prior to using a newly discovered service, fault-detection in the form of regression and online testing, is performed independently of the main application thread. Upon detection of (potential/future) faults, adaptation will be performed, e.g. by rediscovering an alternative service.

Proactive adaptation was also used in [29] to allow physical actions (e.g. turning on or off a fan), to be performed automatically based on the history of user behaviour. Arguably, such an approach is not as promising when applied to the adaptation of interest to this thesis (i.e. application partitioning), since the concerned context (i.e. application behaviour and execution condition) is becoming increasingly dynamic due to the wide adoption of pervasive and mobile computing paradigms. For example, accurately predicting the connection speed of a device involves the consideration of various factors, such as connection type (e.g. wireless), error rate (e.g. dropped packets), distance (e.g. from the access point), or competing processes (e.g. external applications), each of which could change suddenly and drastically due to the characteristics exhibited by the aforementioned computing paradigms. Nevertheless, despite using a reactive approach, the solution presented in this thesis (i.e. the *proposed solution*), considers historical data (as is the case with proactive adaptation) for addressing short-term fluctuations in context changes because such fluctuations could negatively affect adaptation decision making as discussed in section 4.1.4.

Depending on the distribution of adaptation decision making, application adaptation can be categorised into *local adaptation* and *global adaptation*. The local adaptation scheme distributes the decision making responsibility to the individual hosts/nodes in the network, whereas in global adaptation, adaptation decisions are made by a single centralised component. As a consequence, the global adaptation scheme suffers from the issues of having a performance bottleneck as well as a single point of failure. On the other hand, local adaptation generally produces less optimal adaptation decisions due to the limited information available to individual nodes.

For example, existing application partitioning solutions [73] [186] [53], which assume the existence of a *centralised adaptation engine* (i.e. *global adaptation*), perform a specific partitioning technique (e.g. min-cut algorithm [163]), on a graph representing the architecture of the application. In the graph, a vertex represents an application object, whereas an edge represents an invocation relationship between two objects. Each vertex and edge may be weighted based on the resource consumption of the represented object/relationship. For ex-

ample, a weight can be assigned to a vertex to indicate the execution intensity (e.g. CPU consumption) of the object, whereas a weight representing the interaction intensity (e.g. bandwidth consumption) of a pair of objects may be assigned to the relevant edge.

On the other hand, such an approach is not feasible in local adaptation since not only is it costly to exchange detailed object information between *distributed adaptation engines*, the decision making in such adaptation is inherently isolated, i.e. objects are managed by the adaptation engine running on the same node. Furthermore, as discussed in section 2.4, the management of context (which is represented as *metrics*) in such adaptation is not trivial, since the distributed nature of decision making presents questions related to when, where and how metrics are delivered and/or exchanged.

2.4 Context Management

The domains in which *application adaptation* (e.g. [87], [38], [109]) and *context awareness* (e.g. [16], [50], [164]) are applied, often differ, although both areas involve similar operations and goals, which is to address variability of certain entities/attributes (e.g. user location, availability of resources) by altering application behaviour. In particular, application adaptation is traditionally concerned with variability in the utilisation or availability of hardware resources (e.g. processor, network bandwidth) as reviewed in sections 2.1, 2.2, and 2.3, whereas context-aware applications (e.g. [188], [41]) concern information about the user (of the application) and the physical environment in which he/she is located.

Consequently, automation, which is the primary goal of application adaptation, may not be fully achieved in context-aware applications despite being an objective, because some context-aware applications require user-provided information (e.g. user preferences [106] or constraints [41]). For instance, [41] presents a tourist guide application, which is able to navigate a tour group from one attraction to another, based on the user's location (automatically obtained) and time constraints (manually specified). Despite the difference in focus, work on context awareness is nonetheless included in this review, due to its extensiveness in the investigation of issues related to context management, some of which are directly relevant to this thesis, as outlined in the following discussion on the history of *context*.

As surveyed in [16], the term *context aware* was first introduced in [152], which defines *context* as the location and identities of nearby entities (e.g. people and physical objects) as well as changes to those entities. Such a restrictive definition reflects the focus of early work on context-aware systems [188] [152] [2], which considers location (of a person or physical object) as the primary attribute/context for altering application behaviour. For instance, [188] proposes a system capable of determining the closest telephone to a user wearing a location-aware badge, and routing phone calls (targeted at the user) to the telephone.

As studies on context awareness become more pervasive, the term *context* is used to refer to a broader range of information [45] [106] [137] [85] [154]. For example, [137] identifies *logical context* such as user's goals and business processes, as an alternative to the traditionally used *physical context*. A more general definition is used in [85], which considers context as any aspects of the current situation. Similarly, [106] describes context as anything other than the explicit inputs (to a program) that affects the computation and outputs of the program.

[45] provides a more elaborate definition for context, in which three distinct types of entities were considered: places (e.g. rooms, streets), people (e.g. individuals, groups) and things (e.g. physical objects, software components). Each of these entities can be further classified into: identity, location, status, and time, which as a result, allows information, such as hardware resource conditions (e.g. bandwidth availability) and internal application attributes (e.g. object interaction) to be considered as *context* (i.e. under the "status" category). Such information is directly relevant to this thesis since it is required for dynamic/adaptive application partitioning.

In comparison to existing work on application adaptation (e.g. [87], [109]) which generally focuses on how adaptation is performed as opposed to how the required information is collected and managed, work on context-awareness (e.g. [75], [129] [13]) explicitly addresses context management, including context modelling, context measurement, context delivery, and context sharing. Issues that are of particular relevance to this thesis include *context modelling* (e.g. [75]), *context measurement*, and *context delivery*, which refer to how context and its relationships are modelled, how context is obtained, and how it is delivered to the adaptation engine, respectively. On the other hand, *context sharing*, which refers to the task of exchanging context between independent information sources (e.g. application components), is of lesser concern to this thesis, as discussed below.

Context sharing addresses the issue of incompleteness [153], which refers to the inability of a single context source to capture all required context. Context may be shared between different applications (i.e. *inter-application*) or between different components of an application (i.e. *intra-application*). In general, the former scenario deals with heterogeneity since the collaborating applications do not necessarily share common information semantics, communication protocols, etc., whereas this is not so with the latter case because semantics and protocols can be predefined. Inter-application context sharing presents interoperability challenges in several areas of operation, such as the retrieval of context (e.g. [93]), the management of conflicts/contradictions (e.g. [129]), and the maintenance of semantic consistency (e.g. [39]).

Context retrieval was addressed in [93] by using a context management system that is organised as a virtual database which supports SQL-like queries, thus allowing context to be accessed via a well-understood and consistent interface. Similarly, [141] presents a context

query language which satisfies a set of requirements compiled based on the characteristics of context (e.g. static/dynamic, temporal) and context sources (e.g. distributed, mobile). Among the querying supports provided by the language is the ability to incorporate mechanisms/functions for complex filtering and aggregation of context.

Due to the various ways in which context is represented in different applications, it is important to ensure that the semantics of the shared context are consistent. Such a requirement leads to the use of languages/technologies for authoring *ontologies*, such as Resource Description Framework (RDF) (e.g. [102]) or Web Ontology Language (OWL) (e.g. [50]), because as commonly applied in the Semantic Web [19], such technologies facilitate semantic mapping between various vocabularies used by the collaborating components. At the architectural level, the consistency issue is addressed in [39] through the deployment of an intelligent broker, which serves to obtain and manage context from multiple sources, as well as redistribute them to multiple clients.

Conflict management addresses context mismatches [129] [13] [30] [140] [93] [101], which refers to conflicts or even contradictions in context of the same type (e.g. user location) that are gathered from multiple sources, e.g. GPS (Global Positioning System) enabled navigation devices and mobile phones. These are often addressed (e.g. [101]) by attaching quality indicators [72], such as resolution, accuracy/confidence, coverage, etc. to collected context, allowing context to be ranked and chosen/used according to their quality. Such a direction has led to the investigation on how Quality of Context (QoC) can be modelled [101] [175] and measured [98] in previous work.

The aforementioned issues are not applicable to intra-application context sharing, which is the focus of this thesis, since variability in retrieval strategies and context semantics, can be prevented by using predefined or pre-agreed retrieval mechanisms (e.g. via an API), context models (e.g. ontology-based models [164]), and context types (e.g. network bandwidth utilisation). Furthermore, as demonstrated in Chapter 4, context management can be coordinated, so that each component of the application is assigned non-overlapping responsibilities (e.g. collecting different context), thus preventing conflicts.

Note that in this thesis, intra-application context sharing is not an independent task, but rather is part of context delivery, which refers to the process of delivering context from individual context sources to the adaptation engines, as addressed in section 4.1.2. The context sources include those belonging to the application (e.g. application objects or injected proxies) and those belonging to the supporting middleware/framework (e.g. middleware components).

Since the proposed solution targets applications authored in the same programming language (e.g. Java) as the middleware/framework, the use of consistent development paradigms (e.g. object-orientation) and protocols (e.g. Java RMI) can be enforced, thereby simplifying

the solution and its implementation. For example, remote communication can be implemented using the default protocol of Java RMI (i.e. Java Remote Method Protocol) rather than more complex protocols such as CORBA (e.g. RMI-IIOP) or web services (e.g. SOAP), which are required for heterogeneous language-level interoperability.

2.4.1 Context Modelling

Since interoperability has been a major concern in existing work [164] [50] [187] [39] [68], there has been substantial adoption of ontologies for modelling collected context. Since ontology-based modelling also provides support for logic inferencing and knowledge reuse, such an approach was favoured in [164], which also considers five other types of model: key-value models (e.g. [151]), markup scheme models (e.g. [31]), graphical models (e.g. [75]), logic-based models (e.g. [4], [72]), and object-oriented models (e.g. [41], [25]).

The key-value modelling approach [151] uses a key-value pair to model context, in which generally, the key portion represents the type of specific context information, and the value portion represents the collected context or the result of certain computation (e.g. averaging of values collected over time). Due to its simplicity, such an approach is more suitable for applications in which less information (e.g. context metadata, relationships) needs to be maintained.

The markup scheme approach uses markup languages such as XML (Extensible Markup Language), which is useful for storing hierarchically interrelated context information as well as its metadata. An XML-based model benefits from the availability of supporting technologies such as XML schema, as used by the middleware presented in [31] to enforce model validity. Similarly, information retrieval and querying the model is facilitated using XPath expressions.

The graphical approach models various context information and its relationship as a graph, such as that applied in [75], which was based on the ORM (Object Role Modelling) approach. In the solution, the graph is used to define a series of facts (e.g. located at), each of which connects two entities (e.g. a person and a building). A fact will be tagged as “alternative” as opposed to “ordinary”, upon detected conflicts/ambiguity, in order to provide more informed reasoning of context.

Context reasoning is facilitated in [72], by using first-order predicate logic as a formal representation, in order to model collected context in the form of propositions and relations. Note that such an approach is considered as a logic-based modelling approach, even though it is similar to ontology models such as RDF, which uses subject-predicate-object expressions. In fact, according to the classification of context modelling presented in [164], a context

model might fall into multiple categories. For example, an RDF-based model that is expressed in XML format can be considered as ontology and/or markup-based modelling.

In object-oriented modelling [25], context is represented as objects, thereby enabling encapsulation of a specific context (e.g. execution time) and tasks related to the management of the context (e.g. time measurement). Furthermore, such a model allows grouping of context (e.g. time-based context), encourages code reusability, and promotes abstraction through inheritance and polymorphism. According to the survey and evaluation done in [164], the object-oriented modelling approach is favoured over most approaches, with the exception of ontology-based approaches. This is due to weaker support for interoperability in comparison to ontology models, which however as previously mentioned, is not a concern in this thesis.

Note that given enough information being available in the context objects, it should be possible to build different models (e.g. ontologies) based on the object model when necessary. Consequently, since the object-oriented approach has many benefits as described above, and fits naturally into the concerned development paradigm (i.e. object-orientation) as demonstrated in Chapter 4, the object-oriented context management approach is adopted in this thesis.

2.4.2 Context Measurement

Context measurement refers to the activity of obtaining context, which in the case of this thesis, is a specific form of software measurement, because the context of interest is that related to software, be it directly (e.g. response time) or indirectly (e.g. factors affecting response time which include system resource availability).

In software measurement, *metrics* (e.g. response time) are used to provide a measure of the degree to which an entity (e.g. application, component) possesses an attribute (e.g. performance). A *measure* refers to the value obtained by making a measurement, where *measurement* refers to the act of assigning a value to a specific attribute. According to such definitions, which have been commonly used in previous work although not unanimously [62], the semantic difference between *metrics* and *measures* is subtle, and as such the terms are often used interchangeably in this field of software engineering [54]. For simplicity, the term *metrics* will be used in the majority of discussions in this thesis (unless a precise distinction, such as that presented in the next paragraph, is essential), because *measure* by itself is not useful since it is not representative of the measured attribute. Similarly, in high-level discussions, the term *context* might be used in place of *metrics* or *measures*.

Although in software engineering, measures are often obtained in the form of numbers (i.e. quantitative), according to the ISO/IEC 9126 (2001-2004) definition, measures can also be in the form of categories. Such a definition adopts a classification of measurement scales

that was based on the original classification proposed in [162], which consists of four types (from the least to the most informative): *nominal*, *ordinal*, *interval*, and *ratio*.

The nominal scale represents a categorisation through the use of non-quantitative labels, and therefore it only has one empirical relation, i.e. equality, which can be mapped to formal relations "=" and "≠". An example of this scale is that used in [76] to determine whether a specific user has access to a given communication channel, the result of which matches either "=" or "≠".

The ordinal scale represents a categorisation and a rank order. Possible empirical relations include equality and ordering (formal relations "<" and ">"). For example, the context presented in [100] for quantifying the sound intensity of the surroundings was defined on an ordinal scale, with possible values of "silent", "moderate", or "loud".

The interval scale represents a quantitative scale where the difference between two measures of the same scale is empirically meaningful, but not the ratio. Possible empirical relations include equality, ordering, and difference (formal relations "-" and "+"). A familiar example is the quantification of temperature context [100] in the Celsius scale.

The ratio scale represents a quantitative scale where both the difference and ratio between measures are empirically meaningful. Possible empirical relations include equality, ordering, difference and relative difference (formal relations "/" and "*"). All context/measures investigated in this thesis, which include number of invocations, response time, and resource utilisation, fall into this scale category, thereby enabling direct uses of the measures in non-trivial mathematical formulas, such as those presented in Chapter 3.

Metrics of relevance to this thesis will be discussed in Chapter 3, which shows how the required metrics are applied in the proposed adaptation solution. The presented metrics include both *base* and *derived* metrics. Base metrics refer to those that do not depend on other metrics and thus can be obtained via *direct measurement* as addressed in Chapter 4. On the other hand, as demonstrated in Chapter 3, derived metrics are acquired through calculation involving other metrics, which is a process known as *indirect measurement*. Depending on whether the measured attribute changes during the lifetime of the application, metrics can be either *static* or *dynamic*, a distinction which is considered in Chapter 4 to determine whether metrics should be collected *offline* (i.e. prior to application execution) or *online* (i.e. at runtime).

2.4.3 Context Delivery

Since application partitioning involves multiple collaborating machines, the measurement of context is inevitably distributed across multiple machines. Furthermore, in a modular middleware architecture such as that used in this thesis (as discussed in section 2.5.1), context

may be measured in different application/middleware components. Such a scenario complicates the delivery of context from the originating components (i.e. context sources), to the adaptation engines, which are also distributed in the case of local adaptation. Consequently, context delivery in local adaptation, which involves sharing of context between application components (i.e. intra-application), is addressed in Chapter 4.

Context can be shared using a *pull* method (e.g. [112]) in which a client polls a specific component (i.e. context provider) for context, or *push* mechanism (e.g. [100]), where a context provider actively delivers context to the client. A more advanced form of push-based solutions can be achieved by applying the blackboard concept [16], where context providers and clients are decoupled using the publish-subscribe messaging paradigm. Such an approach can be extended further by enabling automatic discovery of context providers [83]. Both *pull* and *push* approaches are applied in this thesis for addressing different situations as discussed further in Chapter 4. However, the blackboard solution is not adopted because in the case where context is exchanged internally between components of the application, loose-coupling is an unnecessary performance overhead.

A hierarchical context delivery solution, which organises collaborating nodes/machines in terms of a tree of clusters, was presented in [112]. In the solution, a cluster pulls context from the immediate child clusters. The acquired context is aggregated at that particular cluster and delivered to the parent cluster upon request. This thesis uses a similar technique, which is applied to application and middleware components as opposed to machine clusters as will be demonstrated in section 4.1.2.

2.5 Mobile Object Frameworks

Many frameworks (e.g. [53] [134] [51] [147] [17] [3] [74] [77], [107]) have been developed to facilitate the development of applications supporting object mobility due to the complexity involved in converting standard application objects into location-aware objects. The complexity is even higher in the case of adaptive applications since adaptable objects (i.e. *mobile objects*) need to monitor their activities in order to obtain context required for adaptation.

Note that the term mobile object is used to refer to an object of the application that can migrate to different machines. A mobile object is remote (i.e. known as a *remote object*), meaning that it is remotely accessible by objects running on different hosts. Depending on the intended level of granularity, not all objects of the application need to be mobile or remote. In practice, it is the role of the application deployer to tag certain objects/classes as *remote* or *mobile*, prior to the deployment of the application.

Another type of remote object is a *stationary object*, which is an object that stays on a single machine but is still accessible remotely (by objects located on different machines). An

example of a stationary object is a database connector object which in general should remain in the same machine as the target database. As another example, Graphical User Interface (GUI) objects might also need to be made stationary depending on the domain-specific use cases of a given application.

Existing frameworks vary in terms of provided functionality (such as adaptation or concurrency management) as well as development transparency (i.e. how much developer involvement is required) as reviewed in sections 2.5.1 and 2.5.2.

2.5.1 Middleware Architecture

Due to the differences in the target applications and domains, existing mobile object frameworks, although focusing on the same problem area (i.e. object mobility), vary in terms of supported capabilities. For instance, JavaParty [134] and JavaSymphony [51] specifically address issues related to the development of parallel object-oriented applications such as asynchronous invocation of remote objects and synchronisation of concurrent operations, however both frameworks lack support for adaptation. Similarly, [191] provides a mechanism to defer object migration until a later point, at which migration can take place with minimal overhead (i.e. less transferred data), which although is beneficial for applications dealing with large amounts of data (e.g. image processing), does not address the criteria (e.g. low resource availability) for initiating object migration in the first place.

FarGo [64] [79] [80] provides preliminary adaptation support through its monitoring subsystem, which allows monitoring of application-related information without mixing application and monitoring logic. However, a working implementation of usable monitoring functionality and adaptation decision making is not provided. A similar limitation also applies to other adaptation frameworks such as MobJeX [147], Mobile Code Toolkit [123], and a CORBA-based framework presented in [96]. The ADAJ framework [53] [52], which is based on JavaParty, provides better adaptation (including monitoring) support. However, as mentioned in section 2.2, the supported adaptation specifically targets parallel applications as opposed to the more conventional sequential applications.

Despite the differences, existing mobile object frameworks share similarities in terms of the supported basic functionality (e.g. location tracking, remote communication, object migration, etc.) and how it is managed by the provided middleware support/component(s). Consequently, in order to provide a background for the discussion of the adaptation and metrics management solutions presented in Chapter 3 and Chapter 4, a general architectural overview of mobile object middleware follows.

In the simplest case, all runtime operations of a mobile-object-based application are managed by a single middleware component running in each participating host/node in the net-

work. Examples of such frameworks include FarGo [79] and JavaParty [134], in which each running application is managed by a component known as a *core* in FarGo or a *LocalJP* in JavaParty. On the other hand, a more modular system design is adopted by the MobJeX framework [147] whereby the runtime operations are executed through collaboration of multiple cohesive components. For instance, host-related tasks are handled by a *host manager* which is also known as a *service* in MobJeX. On the other hand, process-specific operations which include those related to the application(s) running in the process, are handled by a component called a *runtime*. Note that the MobJeX framework includes several other components as discussed in section 4.3, which addresses issues related to the implementation of the metrics management solution proposed in Chapter 4.

The discussion of the solutions proposed in Chapter 3 and Chapter 4 assumes that in order to manage the execution of an application, a mobile object framework should have at least a *host manager* and a *runtime* even if in frameworks such as FarGo and JavaParty, these are only logical functions of a single middleware component. This is done for clarity, since the adaptation solutions discussed in section 3.1 (i.e. the *original solution*) and 3.2 (the *proposed solution*) involve two distinct sets of tasks, i.e. those related to a specific host and those to a specific process. As shown in Figure 2-1, the mobile object system running in each host/node consists of a group of objects that are interconnected in a tree-like structure. At the root of the system is the *host manager*, which has the role of managing all the *runtimes* running on the host in addition to managing certain host-specific operations such as communicating with other hosts. Underneath the host manager is the runtime which is responsible for managing the *remote objects* of the contained application. Note that although MobJeX allows multiple applications to be managed by a single runtime (as discussed further in section 4.3), for simplicity, subsequent discussion assumes one application per runtime.

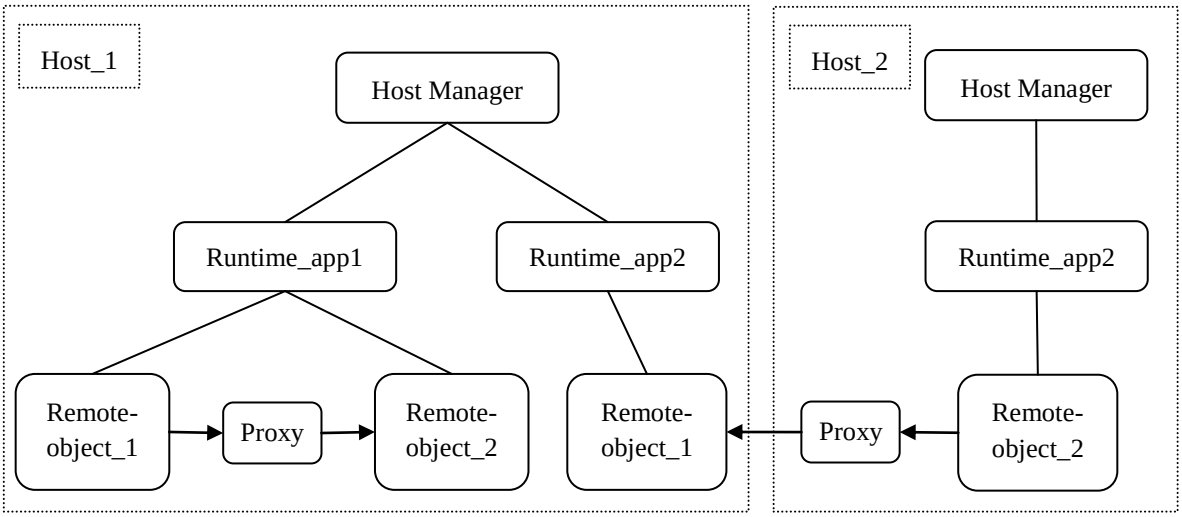

Figure 2-1. An Example of Mobile Object Systems

In existing mobile object frameworks, remote objects are accessed via *proxies* (also known as stubs) as depicted in Figure 2-1. In this work, the term *proxy* is used to refer to an *object-level proxy*, which refers to an application object that intermediates the communication between two objects, as opposed to an application-level proxy, a common example of which is a web proxy. The use of object-level proxies offers various benefits, making it a suitable approach for many domains including object mobility and adaptation as will be discussed in detail in section 2.6.

2.5.2 Development Support

As the main purpose of mobile object frameworks is to reduce application development/deployment effort, its portability, which allows application/framework code to run on heterogeneous machines without modification, is essential. Consequently, Java, including its complementary platforms/technologies (e.g. JVM, RMI), is a popular choice for the development of existing frameworks (e.g. [178] [147] [134] [80]), in which the targeted/supported applications are those written in Java. Nevertheless, in the specific case where interoperability is a main concern, technologies such as CORBA, can be used [86] [96]. However, in this case, the advantage of interoperability comes at the expense of limited supporting tools and application coverage (i.e. since fewer applications are developed with support for CORBA).

In addition to providing middleware support, which was described in section 2.5.1, many mobile object frameworks provide basic development support in the form of an API, through which applications can access functionality and services supported by the framework. Relying on such support alone places a greater burden on application developers since the implementation of adaptive application partitioning via object mobility (which is a type of *application-aware adaptation*) involves authoring of non-trivial functionality such as that which enables location-transparent communication between remote objects.

Consequently, frameworks supporting distributed applications such as FarGo [80], JavaParty [134], Shadows [34] or Arjuna [130], support automatic generation of stubs or proxies. Note that subsequent discussions only focus on Java-based frameworks due to the fundamental differences between various technologies, which include platform portability (e.g. Java does not have issues with byte-ordering) and language constraints (e.g. Java does not support multiple inheritance).

In FarGo, the generated stub/proxy class contains functionality required to locate and communicate with a *complet* (i.e. the smallest migration unit which comprises multiple objects). The main limitation of such an approach is that it introduces a new development model wherein objects are grouped as complets, and thus additional development effort is required for defining complet interfaces and adhering to specific APIs.

On the other hand, frameworks such as JavaParty, J-Orchestra, and MobJeX, provide a more transparent solution in which not only does additional code get generated automatically, but the generated code also gets injected into a normal application (e.g. a standard Java application) with reduced human intervention. The injection is done through a process known as *code transformation*, which involves modifying existing application code. Such a solution is traditionally performed at source-code level however transformation at byte-code level is possible in byte-code oriented languages such as Java. Such flexibility serves as a motivating factor for targeting Java applications (as is the focus of this thesis), since there has been considerable work on Java-based transformation technologies and tools, such as AspectJ [10] and ASM [28].

Among the code that gets injected is code related to *proxies* (of remote objects), which as will be discussed in section 2.6, serve an important role in the realisation of adaptive application partitioning via object mobility. Consequently, proxy injection which refers to the process of automatically inserting proxy code into existing application code (via code transformation) is a major concern in this thesis as elaborated in section 2.6.

2.6 Capability Injection

This section introduces the challenges in capability injection, which in the specific context of this work refers to the injection of capabilities into an ordinary/non-adaptive application for the purpose of converting an application into that capable of adapting via object mobility. Such injection not only eases the development of new applications, since the developer can focus on the application logic (rather than the added/extra capabilities), but also assists the extension of existing applications with reduced development effort. The realisation of such capability injection relies heavily on the properties of proxies (i.e. object-level proxies) as discussed below.

An object proxy, which intermediates the interaction between two objects (i.e. *inter-object* communication), can be used for deferring the creation/initialisation of the target object (i.e. *lazy initialisation*), redirecting the communication target to one or more objects, performing additional capabilities before/after communicating with the target object, etc. As such, proxies have been applied for various purposes, such as memory efficiency (through lazy initialisation) [60], remote communication [173] (including data marshalling/unmarshalling and distributed garbage collection [156]), and concurrent data evaluation/processing (in the form of a *future* object) [136].

In application adaptation, proxies allow applications to dynamically reconfigure themselves (e.g. changing its components, altering its functional behaviour) transparently, without the awareness of the rest of the components in the application as utilised in existing adapta-

tion frameworks, such as ADAJ [52] or MoCA [145]. More specifically, in adaptation achieved via object mobility, proxies serve to provide location transparency [185] to the caller/client objects, allowing the callee/mobile object to migrate independently without affecting the rest of the application. This is because a proxy manages tasks such as tracking/updating the current location of the proxied mobile object and serialising/marshalling invocation messages (i.e. parameters and return values) for remote communication.

Proxies also enable additional processing of invocation messages in order to simulate the *pass-by-reference* behaviour as opposed to *pass-by-copy*, which is the default behaviour for remote communication, as addressed in previous work [147] [176]. Furthermore, certain optimisations can be applied in the proxies so that when the caller and callee are in the same *runtime* (and thus in the same process), the communication is performed via direct/local reference (as opposed to the computationally less efficient inter-process invocation). Proxies also play an important role in facilitating the collection of interaction-related metrics such as invocation frequency as discussed in detail in section 4.1.1.2.

While proxies are normally used to support generalised capabilities (such as those mentioned above) that is common to different classes and applications, the proxies themselves are not generic since each proxy class has to be explicitly authored for a specific class. Consequently, proxy classes, including the contained capabilities (e.g. remote communication), are often automatically generated (at source-code or byte-code level) in order to avoid the routine task of writing repetitive and duplicated functionality. A common limitation in previous work is that even though proxy classes are generated, substantial developer involvement is still required for incorporating the proxies into the application. For example, Java RMI (Remote Method Invocation) [173] and Dynamic Proxy API [170] require the application developers to write an interface consisting of the methods to be proxied prior to the generation of the proxy/stub class.

Similarly, FarGo [79] requires manual authoring of *complet* interfaces as mentioned in section 2.5.2. On the other hand, approaches adopted by frameworks such as EJB 3.0 [171], JavaParty [134], MobJeX [147], and J-Orchestra [178], provide better development transparency by allowing proxied objects to be accessed without requiring a separate interface, thus enabling the application to be written in a manner similar to a normal Java application. The transparency is achieved via *code transformation* whereby proxy-related classes (which contain the to-be-injected capabilities) are automatically generated and incorporated into the original application via automatic modification of existing code. The more compatible the generated/inserted code, in terms of not breaking existing application code (syntactically or semantically), the more transparent the solution is.

However, existing proxy approaches have limitations in terms of the diversity of design and implementation scenarios that can be supported. For instance, EJB 3.0 [171], which uses

proxies to manage the interaction of session beans, lacks inheritance capability between the proxied classes (i.e. session beans). On the other hand, although frameworks such as J-Orchestra [178] allow a proxied class to inherit/extend another proxied class, inheritance involving a non-proxied parent class is not supported, thereby implying a constraint in which a proxied class is not allowed to extend a library class (e.g. a class belonging to the Java API), since as discussed in detail in Chapter 6, it is generally not safe to proxy such a class.

Furthermore, there are other issues affecting proxy transparency such as proxy instantiation, field access, identity semantics, etc., as described in section 6.1, which outlines existing approaches (used to address these issues) as well as the limitations in these approaches. The identified limitations are addressed in section 6.2, which focuses on the transparency of proxy injection in terms of how proxy-related code should be generated and inserted into existing applications without breaking existing code.

On the other hand, issues related to the actual code transformation that is required by the proxy injection solution (proposed in Chapter 6), are addressed in Chapter 7 which focuses on the process and architectural aspects of code transformation as opposed to the specifics of the transformation since tools and technologies for generating and modifying code are adequately available. Examples of such tools/technologies include: 1) specialised program transformation systems such as TXL [43], Stratego/XT [183] [27], and DMS [18], 2) source code transformation tools such as Javacc [89] and Antlr [7], 3) byte-code transformation tools, such as JOIE [42], ASM [28], and Byteman [91], and 4) Aspect Oriented Programming (AOP) tools, such as AspectJ [10].

Note that although proxy injection shares certain similarities with AOP in terms of the ability to inject addition functionality into existing applications, proxies are generally used to bridge inter-object communication rather than intra-object communication which is more appropriately addressed using AOP. Such a distinction differentiates the contribution of this thesis from the majority of work focusing on AOP (e.g. [135], [22]). Furthermore, the focus of this work is not on choosing specific technologies/tools but rather on the adoption of appropriate transformation approaches/techniques, such as source-code versus byte-code transformation and online (e.g. runtime) versus offline (e.g. compile-time) transformation, based on considerations specific to the development and deployment of applications supporting adaptive application partitioning. For instance, even though ideally, no manual authoring of addition code is required in the presence of transparent capability injection, in practice, manual customisation (e.g. extension, fine-tuning) of the injected/produced adaptation functionality might be necessary. Furthermore, the heterogeneity of the target machines (of specific application deployment) should also be taken into consideration as this presents issues related to the compatibility of the produced code as addressed in section 7.1.3. In addition, common

quality attributes such as efficiency, portability, and flexibility, are also considered in the adoption of certain transformation approaches and techniques.

The next chapter (i.e. Chapter 3) proposes an adaptation algorithm for achieving one of the main objectives of this thesis, which is to improve the effectiveness of local adaptation via application partitioning. As discussed in Chapter 3, adaptation effectiveness is quantified by how much the response time of an application can be reduced using adaptive application partitioning.

Chapter 3. Local Adaptation

As discussed in section 2.3, this work focuses on adaptive application partitioning (via object mobility) performed using a local adaptation scheme, in which case, adaptation decision making responsibility is distributed across individual hosts/nodes in the network. An existing solution for local adaptation [144] is used as a basis of the solution presented in this chapter since preliminary evaluation of the base solution has shown promising results. The base solution is hereinafter referred to as the *original solution* (or the *original algorithm* when particularly referring to the adopted decision making algorithm), whereas the algorithm/solution proposed in this chapter is referred to as the *proposed algorithm/solution*.

In the original solution, an adaptation engine, which exists in each participating node/machine, is responsible for adapting mobile objects currently residing in the relevant node. As such, an adaptation engine can be implemented as a sub-component of a *host manager*, which as introduced in section 2.5.1, is responsible for managing activities related to a particular node. When adaptation is triggered (e.g. due to changes in resource availability), the adaptation engine in the adapting (*source*) node makes decisions on which mobile objects to migrate and to which (*target*) nodes the objects should be migrated. The decisions are made based on the characteristics of the adapted objects and the target nodes as well as the goals of the adaptation (e.g. improving the overall response time of the running application).

The characteristics of the mobile objects and the target nodes could be captured by analysing relevant context information (i.e. metrics), the collection of which is the main focus of Chapter 4. On the other hand, the adaptation goals depend on the chosen adaptation algorithm and how it is configured (by the application deployer) as discussed in the next section (i.e. 3.1), which provides an overview of how adaptation decisions are made in the original algorithm [144] while also emphasising certain limitations, which will be addressed in the proposed algorithm (presented in section 3.2). Since adaptation only concerns mobile objects as opposed to other types of objects (e.g. stationary objects), subsequent discussions are mainly aimed at mobile objects, and as such, for simplicity, the term *object* is used to refer to a mobile object unless explicitly stated otherwise.

Both the original and the proposed algorithms presented in sections 3.1 and 3.2 have been implemented in an existing Java-based mobile object framework called MobJeX [147]. However, since the implementation is straight-forward, detailed discussion about the framework will instead be provided in Chapter 4, which concerns the management of metrics required for adaptation. Empirical evaluation concerning the behavioural correctness and the effectiveness (in terms of how much benefit is gained) of the two adaptation algorithms is pre-

sented in Chapter 5 since the latter requires the metrics management solution presented in Chapter 4.

3.1 Original Algorithm

The original algorithm proposed in [144] utilises a scoring system for its adaptation decision making. In this approach, *a score* is assigned to each *pair* of mobile object and potential destination node. The score for the object-node pair reflects the degree of improvement (e.g. application response time) that would be gained by migrating the mobile object to the node in that pair. The more improvement expected from the adaptation/migration, the higher the score is.

Another factor that affects the score is the adaptation *score threshold*, which is preconfigured by the application deployer. The lower the threshold, the higher the calculated score, as can be seen from the pseudo-code of `calculate()` presented in Figure 3-1, which subtracts the specified `threshold` from the calculated value `iScore`. Note that `iScore` does *not* refer to the final score (which is calculated in `evaluate()`), but rather refers to an intermediate *indicator* value which contributes to the final score as will be explained later in this section. One limitation of the original adaptation algorithm is that the specification of the score threshold (i.e. by the application deployer) relies heavily on a trial-and-error approach. This issue will be discussed in more detail in section 3.2.2.4.

The calculation of the score is done in such a way that the resulting score is always between 0 and 1 as shown in `calculate()`. As such, the middle value of 0.5 is used to determine whether a migration should occur, as shown in `adapt()`. The object-node pair producing the highest score (among all pairs) that is above 0.5, will trigger a migration in which the object in the highest-scoring pair will be migrated to the node in that same pair. This behaviour exhibits an important characteristic of the algorithm whereby a mobile object is adapted and migrated independently from other objects. This characteristic could result in less optimal adaption decision making as will be described in detail in section 8.3.

```
adapt() {
   do {
      maxScore = 0
      maxObject = null
      maxNode = null
      for each mobile object m in source node {
         for each target node n {
            score = evaluate(m, n)
            if (score > maxScore) {
               maxScore = score
               maxObject = m
```

```
                maxNode = n
             }
          }
       }
       if (maxScore > 0.5) {
          Move maxObject to maxNode
       }
    } while (maxScore > 0.5);
}

// See Figure 3-2 for the mathematical formula representing evaluate()
evaluate(object, node) {
    totalScore = 0
    for each indicator i {
       if (i.weight > 0) {
          totalScore += i.weight *
              Math.pow(calculate(i, threshold, object, node), power))
       }
    }
    return Math.pow(totalScore, 1 / power)
}

calculate(indicator, threshold, object, node) {
    // Figure 3-3 shows how the score of performance indicator is calculated
    iScore = indicator.calculateScore(object, node)
    // Ensure that the returned score is between 0 and 1
    // maxIndicatorScore refers to the maximum possible value returned by
    // indicator.calculateScore()
    return (0.5 + (0.5 * ((iScore - threshold) / (2 * maxIndicatorScore))));
}
```

Figure 3-1. General Control Flow of Local Adaptation Algorithm

After the migration of a particular object, the score calculation process is repeated until none of the scores in that iteration is above 0.5. Strictly speaking, this *score recalculation* is not required since none of the computation variables (e.g. metrics) used in the calculation have changed at any point in the adaptation. The *recalculation* was in fact introduced in the original algorithm [144] for future use, which as will be discussed in section 3.2.1 is essential for the proposed algorithm.

With regard to adaptation goals, the original algorithm proposes different types of improvement that can be expected/gained from an adaptation/migration, which include resource usage balancing (e.g. processor, memory, and network) and application performance (e.g. response time). In the pseudo-code (Figure 3-1) and the main formula (Figure 3-2), these improvement types are referred to as *indicators*. Each *indicator* is calculated in a different way (e.g. using different metrics), but multiple indicators can be considered in the score calculation. The importance of the individual indicators can be adjusted by assigning a certain

weight to each indicator as can be seen in `evaluate()` in Figure 3-1 and in the mathematical formula (representing `evaluate()`) in Figure 3-2.

$$Score = (W_{RT} I^{P}{}_{RT} + W_{PU} I^{P}{}_{PU} + W_{NU} I^{P}{}_{NU} + W_{MU} I^{P}{}_{MU})^{1/P}$$

Where:
W = Weight of each indicator
I = $IndicatorScore = 0.5 + 0.5\,(d - t)/(2 \times max)$
  d = Raw difference = significance in improvement (see Figure 3-3)
  t = Threshold
  max = Maximum possible value of d
P = Weight power
RT = Response time improvement
PU = Processor usage balancing
NU = Network usage balancing
MU = Memory usage balancing

Figure 3-2. The Main Local Adaptation Formula

The `power` variable shown in `evaluate()` and in the formula, provides a more flexible/sophisticated way of tuning the relative weights between indicators by the means of a non-linear *weighted power mean* technique as explained in detail in the original work [144]. However, since this thesis focuses solely on one indicator, which is on application performance (I_{RT}), the weight and power are unnecessary and thus are not discussed further.

I_{RT} is chosen as the main focus of this thesis because amongst the four indicators considered in the formula (Figure 3-3), the calculation of I_{RT}, which will be covered in section 3.1 and 3.2, is the most complex. Furthermore, the core metrics required for calculating the resource usage balancing indicators (i.e. I_{PU}, I_{NU}, and I_{MU}) are subsets of the metrics used in I_{RT}. This is especially the case in the proposed algorithm which uses a more involved set of metrics than the original algorithm. Consequently, this and the remaining sections will only discuss the calculation of performance scores, which in the specific case of the original algorithm is defined as the application *response time*. Section 3.1.1 provides a detailed discussion of the scoring formula, whereas section 3.1.2 shows how individual metrics apply to the formula.

3.1.1 Performance Prediction Formula

As mentioned in section 3.1, the adaptation/migration of a mobile object occurs independently from other mobile objects, thus the scoring for application performance (i.e. response time) only uses variables/metrics specific to the mobile object being adapted. This is done with the assumption that other aspects of the application (including the location of other objects) remain the same, which is correct at the point when the scoring is being calculated.

* See Table 3-1 for information on the required metrics.

(1) $PerformanceDifference = ORTsrc - ORTdst - MT$

```
Where:
src = Source machine (before adaptation/migration)
dst = Destination machine (after adaptation/migration)
```

(1.1) $ORTsrc = \sum_{i=1}^{Methods} MRTsrc_i \times NI_i$

```
Where:
Xi = The metric X that is measured during the invocation of method i
```

(1.2) $ORTdst = \sum_{i=1}^{Methods} MRTdst_i \times NI_i$

$$= \sum_{i=1}^{Methods} (ETdst_i + ITdst_i) \times NI_i$$

$$= \sum_{i=1}^{Methods} ((ETsrc_i \times \frac{PAsrc}{PAdst}) + ((MRTsrc_i - ETsrc_i) \times \frac{NAsrc}{NAdst})) \times NI_i$$

(1.3) $MT = MIT + MCT$

```
Note: MT was introduced as a concept but was not implemented
```

Figure 3-3. Performance Prediction Formula in the Original Algorithm

In formula 1 in Figure 3-3, calculating the performance score for a object-node pair involves three main computation components: 1) *ORTsrc* (Object Response Time in source node), which refers to the *response time* of the mobile object when it is in the source node, i.e. prior to migration; 2) *ORTdst*, which refers to the predicted/computed response time when the mobile object is in the destination node, i.e. after migration; and 3) *MT* (Migration Time), which refers to the time required to migrate the object to the destination node. Table 3-1 lists all the metrics that are required by the original formula (Figure 3-3). Some metrics have been named differently from the original version so that they are consistent and distinguishable from the additional metrics introduced in the proposed algorithm presented in section 3.2.

Metrics		Description	Unit
Scoring	Object Response Time (ORT)	The time required to invoke and execute all methods of a mobile object	ms
	Migration Time (MT)	The total time required to migrate a object	ms
Soft-	Method Response Time (MRT)	The time required to invoke and execute a method	ms
	Execution Time (ET)	The time required to execute a method body	ms

	Invocation Time (IT)	The time required for invoking a method (excluding the ET portion)	ms
	Number of Invocations (NI)	The number of times a method is invoked (and thus executed)	integer
	Migrate Instance Time (MIT)	The time required to migrate an object	ms
	Migrate Class Time (MCT)	The time required to migrate the class of an object and other classes referenced by the class	ms
Resource	Processor Availability (PA)	The processor/computing power of a host/node that is available to the application	instructions/s
	Network Availability (NA)	The network bandwidth of a host that is available to the application	bytes/s
	Host Processor Capacity (HPC)	The processor capacity of a host	instructions/s
	Host Network Capacity (HNC)	The bandwidth limit of a host	bytes/s
	Host Processor Usage (HPU)	The processor utilisation of a host	instructions/s
	Host Network Usage (HNU)	The network utilisation of a host	bytes/s

Table 3-1. Metrics Required by the Original Formula in Figure 3-3

In the formula (Figure 3-3), the *ORTdst* computed in equation 1.2 is subtracted from the *ORTsrc* computed in equation 1.1, to determine how much ORT would decrease (i.e. performance would improve) had the mobile object been in the destination node instead. The larger the difference, the more the performance improvement gained from the adaptation/migration. On the other hand, a negative subtraction result indicates that the object (or the application as a whole) will fare worse in the destination node. This would happen if the destination node had lower availability of certain resources, such as processor time or network bandwidth. On the other hand, a zero result means that there is no difference in resource availability between the two nodes, and thus there is no incentive for migration. In fact, migrating the object in this case will slow down the application execution due to the migration overhead (i.e. extra operations involved in migrating an object), which has also been considered in the formula (sub-formula 1.3).

A detailed explanation of sub-formulas 1.1, 1.2, and 1.3 will be provided in section 3.1.2, wherein the discussion will also cover how certain metrics fit into the sub-formulas. The required metrics (presented in Table 3-1) are grouped into 3 categories: *scoring*, *software*, and *resource*. The *scoring* category refers to conceptual metrics that are used only for the purpose of presenting the scoring formula. The *software* category refers to metrics that capture the application behaviour/characteristics and thus should be measured in the application itself, as discussed further in Chapter 4. Lastly, the *resource* category refers to resource-related metrics

which are essentially environmental metrics, and hence can be measured independently from the application, e.g. by a middleware/framework or the operating system (OS).

3.1.2 Performance Prediction Metrics

This detailed discussion of the sub-formulas in Figure 3-3 is structured as follows. Section 3.1.2.1 discusses the general concept and the metrics that are involved in the calculation of ORT (Object Response Time). Section 3.1.2.2 explains how the general ORT calculation can be extended to predict the after-migration *ORTdst*. Finally, section 3.1.2.3 describes the calculation of the object migration overhead/time. In the discussions, certain limitations of the original algorithm, which provide a basis for the extensions and improvements presented in section 3.2, will be outlined along with the specific sections where they will be addressed.

3.1.2.1 Calculating ORT

As shown in equation 1.1 in Figure 3-3 (section 3.1.1), which concerns the before-adaptation calculation (i.e. in the source node), the response time of the adapted mobile object (i.e. ORT) is defined as an aggregate of the time required to invoke and execute each method (i.e. Method Response Time) of the object. On the other hand, equation 1.2 calculates/predicts the after-adaptation ORT (i.e. in the destination node), by breaking down Method Response Time (MRT) into Execution Time (ET) and Invocation Time (IT) metrics. This is because as opposed to equation 1.1 which can simply use metrics that were collected prior to the adaptation, the prediction in equation 1.2 requires finer-grained computation as discussed in section 3.1.2.2.

ET refers to the time taken to execute the body/operations of a method, whereas IT refers to the time required to call/invoke the method but excluding the time spent on the execution of the method body. The reason for separating ET and IT is to split apart the resource requirements of the two actions, in which ET is primarily associated with the processor consumption of the method execution whereas IT is primarily associated with the network bandwidth usage of the invocation. However, one limitation of this approach is that the ET and IT metrics are not accurate representatives of processor and network consumption respectively. More specifically, ET does not differentiate between CPU-based and I/O-related operations (e.g. disk accesses), and therefore a long ET does not necessarily imply a CPU-intensive method execution. Furthermore, ET is influenced by external factors such as the load of the machine which is affected by external programs or processes.

Similarly, IT is also affected by the present condition of the network, e.g. the currently available bandwidth. Other limitations of IT are due to the significant differences in the collected measures between *local* and *remote* invocations. More specifically, in a local method invocation situation, in which the caller and the callee objects are in the same process or run-

time (e.g. Java Virtual Machine), IT is insignificant, i.e. very close to 0 milliseconds, because the invocation messages (i.e. parameters and return values) are passed by reference within a single process.

In contrast, the IT measure of a remote invocation could be significant due to the overhead of passing a complete copy of the invocation messages over a slower communication channel (i.e. inter-process versus intra-process communication). This overhead is of particular concern in this problem domain since the executed adaptation (i.e. object migration) could result in two objects being in different nodes (thus different processes) thereby requiring inter-process communications between the objects. However, due to the absence of certain interaction information in the original adaptation solution, the co-locality (i.e. local versus remote) of a method call cannot be determined accurately. Consequently, this issue will be addressed in the proposed solution presented in section 3.2 which requires the collection of IT at a finer granularity (as addressed in section 4.2.1.3).

In addition, since local IT measures (i.e. the ITs of local invocations) are always very close to 0 milliseconds due to the minimal overhead of reference passing, remote ITs cannot be accurately derived from local ITs, thus affecting the prediction of *ORTdst* from sub-formula 1.2 of Figure 3-3. This is of particular concern when all objects are located in the same process, and thus all of the collected IT measures are for local invocations (i.e. roughly 0 ms). Such a case usually happens in the first adaptation execution since all objects initially execute on a single machine (i.e. source machine), unless offline application partitioning (e.g. [186]) is performed prior to application start-up. The aforementioned issues effectively impact on the quality (and thus effectiveness) of the adaptation decision making and as a consequence, more accurate metrics are proposed in section 3.2.2.1.

Due to the difficulty of accurately measuring IT [61], it is presented in the formula as a concrete measure, but in practice, it is calculated by subtracting ET from MRT which are both directly measured. However, since it was not the aim of the work on the original adaptation algorithm [144] to address the collection of the required metrics, certain metric-related aspects were left unaddressed. For instance, the work did not consider in what form the measures (e.g. ET) that are collected during the lifetime of the application should be presented to the adaptation engine. A solution might for example consider presenting only the last collected measure or presenting the average of collected measures depending on the considered benefits/trade-offs as addressed in section 4.1.4. Additionally, other issues, such as the feasibility and the efficiency of metrics collection tasks, also need to be addressed (as is the focus of Chapter 4) in order to enable automated/live adaptation (i.e. using real metrics).

As can be seen from sub-formulas 1.1 and 1.2, ET and IT are multiplied with NI (Number of Invocations) to show the significance of a method/object based on how many times it is invoked or executed. One potential issue with this approach is that it might undesira-

bly/incorrectly favour (i.e. place higher scores upon) a small group of objects for a long period of time. In the case where an object executes frequently at the start of the application execution, it might take some time before other objects can come to the same level of significance. Furthermore, this behaviour also indicates that more preference/importance is placed on objects with longer life spans as the value of NI accumulates. Since this limitation is caused by the lack of temporal information in the measured NI, an alternative metric which incorporates temporality/periodicity will be presented in section 3.2.2.1.

3.1.2.2 Predicting ORT for the Destination Node

As mentioned, calculating *ORTdst* is more complicated than calculating *ORTsrc*, which simply requires software metrics (i.e. MRT and NI) that were measured prior to the adaptation. This is because metrics concerning the adapted object were collected in the source node, thus representing the execution in the source node, but not the destination node. Consequently, in order to predict ORTdst, the original metrics (i.e. ETsrc and ITsrc) are computed against the ratio of the resource availability between the source and destination nodes. The ratio serves as a multiplier showing which of the two machines provides computing/network resources that are more suitable for the object, i.e. will allow shorter response time. As shown in sub-formula 1.2 in Figure 3-3, ETsrc is multiplied with the PA ratio whereas ITsrc is multiplied against the NA ratio.

$$PA = HPC - HPU$$
$$NA = HNC - HNU$$

Where:

PA = Processor Availability	NA = Network Availability
HPC = Host Processor Capacity	HNC = Host Network Capacity
HPU = Host Processor Usage	HNU = Host Network Usage

For more information regarding the involved metrics, refer to Table 3-1.

Figure 3-4. Resource Availability Definition According to the Original Formula

In this formula, resource availability is defined as the difference between the resource capacity and the current usage of the machine as shown in Figure 3-4. According to the definition, resource availability in either machine (i.e. source or target) refers to the *additional* resource as opposed to the *actual* amount that the machine can offer to the application. The former is generally lower than the latter because it does not include the amount that is being used by the application itself (i.e. *internal load*), which in fact also contributes to the amount that the application can utilise. Figure 3-5 provides an example in which the original algorithm perceives that the availability (of a particular resource, e.g. CPU) between two machines (i.e. M1 and M2) is the same, but in reality M1 has higher availability because, due to the nature of a sequential application (which is the focus of this thesis), an object can execute with minimal contention from the internal load (e.g. from other objects of the same applica-

tion), thereby implying that internal load/usage contributes to the resource available to the application (as shown in the depiction of "Real Availability" in Figure 3-5). Note that resource contention from the internal load should be minimal even though sequential applications might consist of multiple threads (as mentioned in section 2.2), because most of the threads are just supporting threads which rarely execute concurrently with the main thread.

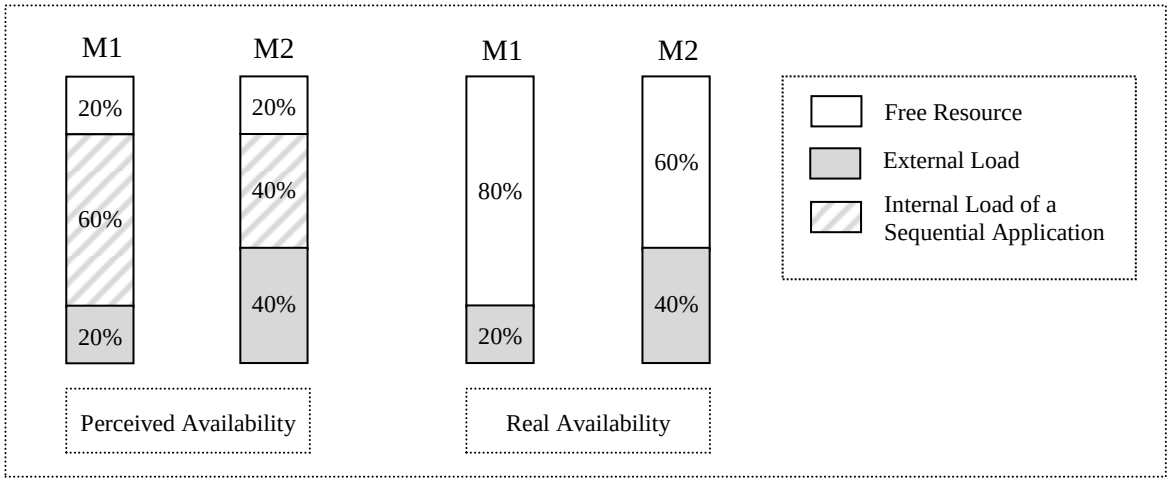

Figure 3-5. Resource Availability in Original Algorithm

The inaccuracy in the resource availability calculation of the original algorithm often causes incorrect migration decisions as illustrated in Figure 3-6, which provides a simple example wherein three mobile objects in an application (i.e. *O1*, *O2*, and *O3*) execute one after another in a specified order. For instance, O1 executes at T1 and O2 executes at T2 wherein each T refers to a specific time slot/unit. The objects have been distributed in such a way that O1 and O3 are located in the source node whereas O2 executes in the destination machine.

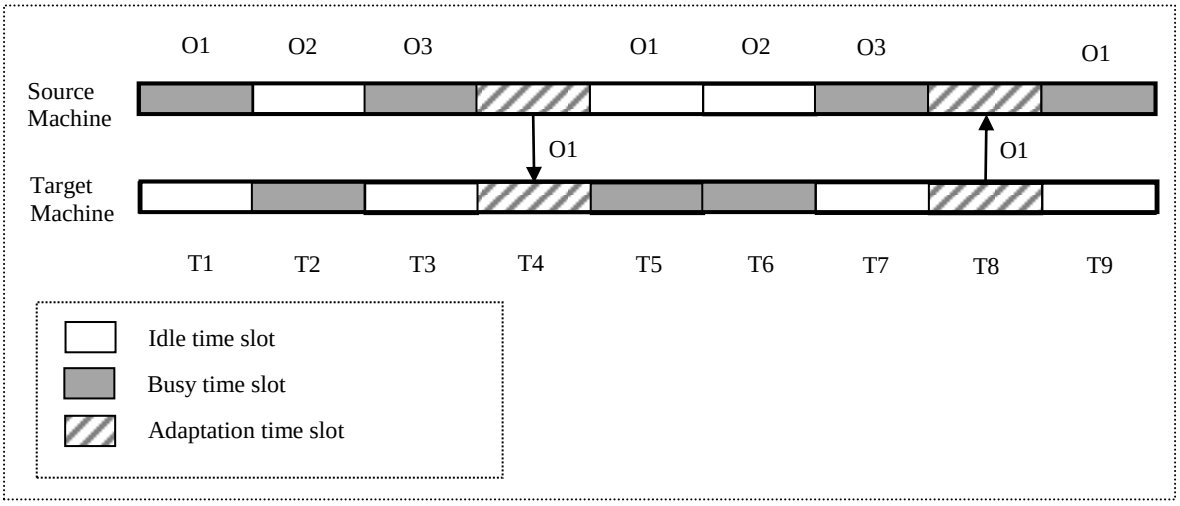

Figure 3-6. An Example of the Ping-pong Phenomenon

For simplicity, the following assumptions are made. The resource (e.g. CPU) capacity of the two machines is assumed to be equal. No other threads or processes are running or competing for resources. The execution of each object is assumed to consume the same amount of

resources. Finally, the measurement of the resource availability is assumed to be performed at every 3T. Consequently, in the adaptation executed at T4, the source machine is perceived to have ~33% availability as opposed to the 66% of the target machine. As such, due to the perceived lower availability of the source machine, O1 would be migrated to the target node. Since O1 would execute in the target node (thus consuming the resource of the node) at T5, the adaptation at T8 would move the object back to the source machine.

This is known as the *ping-pong phenomenon* wherein objects get offloaded back and forth between two machines. Such migrations do not offer any benefit because the source and target machines actually have the same capacity and no other processes are competing for the resource, thus both machines indeed have the same amount of resources if the usage of the application being migrated is considered. In all likelihood, this behaviour will actually result in the degradation of application performance due to the overhead of the involved migrations.

Consequently, resource availability is redefined in section 3.2.2.2 to use additional metrics. Note that the described ping-pong phenomenon, which was explained using a simplistic scenario concerning two machines with similar resource availability, is a specific case of *unnecessary migration*, which refers to a state where objects are migrated unnecessarily to other machines, but not necessarily back-and-forth between two machines. Another factor causing such occurrences is when adaptation is overly *agile*, which refers to the characteristic where an application adapts as soon as certain changes (e.g. in the execution environment) are detected. These changes might in fact be short-term, thus rendering the migration unnecessary. Such an issue is addressed in section 4.1.4 through the use of metrics representation, which applies an averaging technique to moderate the *agility* of adaptation.

Occurrences of unnecessary migration, especially when the original adaptation algorithm is used, can be seen in experiments concerning live adaptation (i.e. using real metrics), which will be presented in section 5.3.

3.1.2.3 Calculating Migration Time

Sub-formula 1.3 presents the calculation of the migration overhead/time of a mobile object as a sum of MIT (Migrate Instance Time) and MCT (Migrate Class Time). MIT refers to the time required to migrate the object itself, where MCT refers to the time required to migrate the classes of the object (i.e. Java byte code). This sub-formula was only presented as a concept, but was not implemented/evaluated in the original work [144] and as such its feasibility, especially in terms of how MCT can be obtained accurately requires further investigation as is described in section 3.2.2.3.

3.2 Proposed Algorithm

This section presents the *proposed* local-adaptation algorithm which forms the main contribution of this chapter. As is the case with the *original* algorithm proposed in [144], the adaptation engine in the adapting/source node iterates through all mobile objects in the node as well as all potential destination nodes to find the object-node pair with the highest score. The selected pair defines the object that should migrate and the destination (node) of the migration.

The most notable improvement of the proposed algorithm is the quality/effectiveness of the adaptation decision making as shown in the experimentation results in section 5.3. As discussed in more detail in sections 3.2.1 and 3.2.2, the improvement is achieved by modifying the algorithm to cater for: 1) the substitution of certain metrics (e.g. Number of Invocations) of the original algorithm with more accurate alternatives (e.g. Invocation Frequency), 2) the introduction of additional metrics (e.g. Runtime Processor Usage) to address specific limitations of the original algorithm (e.g. unnecessary object migrations), and 3) the incorporation of more comprehensive information (e.g. interaction between individual objects) into the adopted metrics to allow more precise predictions.

A minor improvement is also incorporated into the proposed algorithm to prevent the migration of a mobile object to a specific machine or runtime if the machine/runtime does not have sufficient memory. In the case where objects are migrated to memory-constrained machines, the performance of the relevant application will degrade due to page faults. On the other hand, when object migration involves runtimes allocated with limited memory, application execution will likely halt due to failures. The memory availability (MA) checking requires the collection of various metrics, which include Object Memory Size (OMS), Runtime Memory Capacity (RMC), and Host Memory Capacity (HMC), as considered in the metrics management solution proposed in Chapter 4.

The checking is performed prior to the calculation of the adaptation scores discussed in sections 3.2.1 and 3.2.2 to prevent unnecessary calculation (when insufficient memory is detected). For clarity, the discussion of the proposed score calculation is structured as follows. Section 3.2.1 presents the basic improvement/extension using metrics that are directly comparable to those used in the original algorithm to outline certain differences between the algorithms. Section 3.2.2 further elaborates the proposed algorithm by introducing additional metrics into the formula.

3.2.1 Performance Prediction Formula

As shown in Figure 3-7, the proposed algorithm improves upon the original algorithm by redefining the performance of an object as ORUI (Object Resource Usage Intensity), which re-

fers to the degree to which an object spends its time performing operations that require hardware resources such as CPU cycles and network bandwidth. The main motivation for the redefinition is that the original definition, i.e. ORT (Object Response Time), suffers the same limitation as NI (Number of Invocations) which tends to undesirably/incorrectly assign higher scores to objects with certain characteristics, e.g. objects with a long life span. This limitation is due to the fact that ORT/NI does not reflect the temporality/periodicity of the captured application behaviour as mentioned in section 3.1.2.1. On the other hand, ORUI which uses IF (Invocation Frequency) instead of NI, does not suffer from this limitation due to the advantages offered by IF as will be discussed in section 3.2.2.1.

A further advantage of using ORUI is that due to its more accurate representation of the resource usage intensity of the object, the configuration of adaptation-related thresholds is simplified as will be discussed further in 3.2.2.4. Table 3-2 presents the core metrics that are used in the proposed formula, the role of which will be discussed in detail in section 3.2.2.

* See Table 3-2 for information on the required metrics.

(1) $PerformanceDifference = ORUIsrc - ORUIdst - MC$

Where:
src = Source machine (before adaptation/migration)
dst = Destination machine (after adaptation/migration)

(1.1) $ORUIsrc = \sum_{i=1}^{Methods} MRUIsrc_i\ outgoing + MRUIsrc_i\ incoming$

Where:

(1.1.1) $MRUIsrc_i\ outgoing = (\sum_{j=1}^{RCallees\_src} IF_{ij} \times MTTsrc_{ij})$

(1.1.2) $MRUIsrc_i\ incoming = (\sum_{j=1}^{RCallers\_src} IF_{ij} \times MTTsrc_{ij}) + (IF_i \times MPTsrc_i)$

Where:
RCallees_src = *Callee* objects that are remote to the adapted object, i.e. located in a node other than the *source* node
RCallers_src = *Caller* objects that are remote to the adapted object, i.e. located in a node other than the *source* node
X_{ij} = The metric X that is measured during the invocation/interaction between method i (either as a caller or a callee) and object j
X_i = The metric X that is measured during the interaction between method i and any object

(1.2) $ORUIdst = \sum_{i=1}^{Methods} MRUIdst_i\ outgoing + MRUIdst_i\ incoming$

Where:

$$(1.2.1)\ MRUIdst_i\ outgoing = \left(\sum_{j=1}^{RCallees\_dst} IF_{ij} \times MTTdst_{ij}\right)$$

$$= \left(\sum_{j=1}^{RCallees\_dst} IF_{ij} \times MTTsrc_{ij} \times \frac{NAsrc}{NAdst}\right)$$

$$(1.2.2)\ MRUIdst_i\ incoming = \left(\sum_{j=1}^{RCallers\_dst} IF_{ij} \times MTTdst_{ij}\right) + (IF_i \times MPTdst_i)$$

$$= \left(\sum_{j=1}^{RCallers\_dst} IF_{ij} \times MTTsrc_{ij} \times \frac{NAsrc}{NAdst}\right) + (IF_i \times MPTsrc_i \times \frac{PAsrc}{PAdst})$$

Where:
RCallees_dst = *Callee* objects that are remote to the migrated object, i.e. located in a node other than the *destination* node
RCallers_dst = *Caller* objects that are remote to the migrated object, i.e. located in a node other than the *destination* node

$$(1.3)\ MC = \log MIT$$

Figure 3-7. Performance Prediction Formula in the Proposed Algorithm

As is the case with ORT, ORUI is calculated with regard to resource availability which means that the same mobile object would have a different ORUI value if it was placed in a machine with different resource availability. A higher ORUI value indicates that an object has to spend a longer period of time to complete operations that utilise certain resources (i.e. processor or network) of the machine. ORUI is calculated through the aggregation of the resource utilisation intensity of each method of the object, i.e. MRUI (Method Resource Usage Intensity).

	Metrics	Description	Unit
Scoring	Object Resource Usage Intensity (ORUI)	Aggregate resource utilisation intensity for all methods of a mobile object.	integer
	Method Resource Usage Intensity (MRUI)	The degree to which a method spends its time to perform CPU/network-bound operations. Variations include: outgoing and incoming MRUI.	integer
	Migration Cost (MC)	The cost of migrating a mobile object.	integer
Software	Invocation Frequency (IF)	The frequency at which a method is invoked.	integer/ms
	Method Transfer Time (MTT)	The duration it takes to execute the network-bound operations of a method.	ms
	Method Process Time (MPT)	The duration it takes to execute the processor-bound operations of a method.	ms
	Migrate Instance Time (MIT)	The time required to migrate an object.	ms

Resource	Processor Availability (PA)	The CPU/computing power available for an application to execute the methods of an object.	instructions/ms
	Network Availability (NA)	The bandwidth available for the communication between two remote objects.	bytes/ms

Table 3-2. Core Metrics Required by Proposed Algorithm (Figure 3-7)

A further extension to the original algorithm is the decomposition of the formulas (Figure 3-7) into the sub-formulas for calculating MRUI for *outgoing* calls and *incoming* calls as shown in equations 1.1 and 1.2. This extension helps produce better adaptation decisions in comparison to the original formula which only considers incoming calls. Note that the following discussion is presented interchangeably between method-level (i.e. MRUI) and object-level (i.e. ORUI) operations depending on the context of the discussion.

The calculations for *outgoing* and *incoming* calls are done separately because not only do incoming calls reflect the degree of interaction (i.e. *interaction intensity*) between the caller and the object being adapted, but they also affect its *execution intensity*, which refers to how frequently and how long the adapted object (i.e. the methods of the object) executes. On the other hand, outgoing calls only affect the interaction intensity of the mobile object. An object with intensive method executions, i.e. high execution frequency and long processor usage time, will produce a higher ORUI especially when the processor availability is low. On a similar note, if the object performs intensive remote interactions, i.e. high invocation frequency and long network usage time, its ORUI will be heavily dependent on the availability of the network resource. On the other hand, since local interactions do not have significant contributions to the overall application performance, especially when compared to remote calls, they are omitted from sub-formulas 1.1.1, 1.1.2, 1.2.1, and 1.2.2 for the calculation of interaction intensity.

Due to the difference in how *local* and *remote* invocations affect the resource utilisation intensity (i.e. MRUI and ORUI), a separation between the two types of invocation is required. This is achieved by keeping track of the *caller* or the *callee* of the adapted object for each recorded method invocation. The co-locality of the invocation (i.e. local or remote) is then determined based on the location of the adapted object and the location of its caller or callee. In order to compute the *interaction intensity* of a method, the interaction metrics of those invocations that have the same co-locality (i.e. from/to the same machine) should be summed/aggregated, as shown in sub-formulas 1.1.1 and 1.2.1 for callee's co-locality (i.e. outgoing calls) and sub-formulas 1.1.2 and 1.2.2 for caller's co-locality (i.e. incoming calls).

It is important to note that since co-locality is determined based on the current location of the object relative to the caller/callee, the determined co-locality might be different for the

calculation of the *before-adaptation* scenario (i.e. the object is in the source/original machine) and the *after-adaptation* scenario (i.e. the object gets migrated to the target machine). Consequently, the two scenarios have to be treated separately when calculating the interaction intensity of an object. As shown in sub-formulas 1.1.1 and 1.2.1, *RCallees_src* and *RCallees_dst* are used to differentiate between the remote callees in the two scenarios. Similarly, *RCallers_src* (1.1.2) and *RCallers_dst* (1.2.2) are used to refer to the potentially different sets of remote callers in the two scenarios. Note however that the differentiation does not apply to the calculation of execution intensity since it does not need to distinguish between local and remote invocations (sub-formulas 1.1.2 and 1.2.2).

On a similar note, after a successful migration of a mobile object to a specific node, the adaptation scores for the other mobile objects have to be recomputed, because it is likely that the change in the application object layout also changes the interaction intensity of the objects. This *score recalculation* is already supported in the original algorithm as mentioned in section 3.1 and shown in the pseudo-code in Figure 3-1.

The overhead of migrating a mobile object in the original algorithm is defined as the time required to migrate the object, which is not applicable to the proposed algorithm since performance is not quantified in terms of *time* (i.e. ORUI instead of ORT). Instead, the migration overhead is defined as a cost which is calculated based on migration time as will be discussed further in section 3.2.2.3.

3.2.2 Performance Prediction Metrics

This section provides a discussion of how certain metrics (listed in Table 3-2) are used in the proposed algorithm/formula (shown in Figure 3-7) to facilitate more informed and thus more effective adaptation decision making. Additionally, a revised version of the proposed formula, which includes additional metrics, is also included as shown in Figure 3-9.

For uniformity, all time-related metrics are assumed to be in milliseconds, which should be sufficiently precise for quantifying any relevant attributes/information. In practice, some metrics, such as network bandwidth usage (per time unit), cannot be measured at that level of detail and are therefore measured in seconds as will be described in section 4.3.1, which addresses the implementation issues related to the collection of the required metrics.

3.2.2.1 Calculating ORUI

As mentioned in section 3.2.1, the resource utilisation intensity of a mobile object (i.e. ORUI) is the sum of the resource utilisation intensity of all the methods (i.e. MRUI) of the object. Two essential components for calculating MRUI are the Method Process Time (MPT) and Method Transfer Time (MTT) metrics, which refer to the time a method spends executing operations that require CPU cycles (i.e. processing data) and network bandwidth (i.e. trans-

ferring data), respectively. MPT and MTT are more accurate than their counterparts in the original algorithm, i.e. ET (Execution Time) and IT (Invocation Time), because the measured values of ET and IT are contaminated by several factors, such as operations involving unrelated resources (e.g. disk accesses) and external applications competing for the same resource (i.e. processor or network).

Both MPT and MTT are abstract metrics that are presented in the formula (Figure 3-7) to improve clarity/readability as well as simplify the related discussion, but in practice these metrics are derived from other metrics as explained in the discussion that follows. Metrics related to MPT, which are of particular interest to this work, include PUT (Processor Usage Time) and NEI (Number of Executed Instructions). In the context of method executions, PUT refers to the total time a method spends using the CPU of the machine. The problem with PUT is that a PUT measure is only meaningful in the context of a specific machine, i.e. when the processor capacity of the machine is known. As such, there is no guarantee that a method with a certain PUT measure will have the same measure on a different machine even when its behaviour on the two machines is consistent.

On the other hand, this thesis argues that NEI (Number of Executed Instructions) which refers to number of instructions (e.g. Java instructions) that are executed by a specific method, is more universal than PUT and therefore is a more suitable metric for deriving MPT. NEI was first introduced in [146] to support the discussion of the presented preliminary work on adaptation via object mobility. However, as the adopted adaptation algorithm favoured the use of ET, the discussion of how NEI could be used in the algorithm was not provided.

Directly measuring NEI is impractical since it either requires potentially inaccurate code analysis or possibly platform-dependent low-level instrumentation/profiling which are both difficult to achieve due to the involved complexity. Consequently, the NEI metric is instead derived from both PUT and the processor capacity of the source machine (HPC), i.e. the machine where PUT was measured/collected, as shown in Figure 3-8, which also includes the complete steps for deriving MPT, as well as the relationships between other metrics as will be discussed later in the section. Information regarding the new/additional metrics (e.g. PUT) introduced in this section is presented in Table 3-3. The final/complete version of formula, which makes use of the additional metrics, is presented in Figure 3-9.

* See Table 3-2 and Table 3-3 for more information on the involved metrics.

1) $NEI = PUTsrc \times HPCsrc$

NEI = Number of Executed Instructions
PUT = Processor Usage Time
HPC = Host Processor Capacity

- The tag 'src' means that the relevant metric represents a certain attribute in the context of the source machine, whereas 'dst' is used for the destination machine.
- The value of NEI is universal and it can only be derived from metrics that are measured in the source machine since the metrics for the destination machine will not be available until after the object gets migrated.

2) $IF = \frac{NI}{MD}$

IF = Invocation Frequency
NI = Number of Invocation
MD = Measurement Duration

- The value of IF is not affected by the availability of any resources and therefore the distinction between 'src' and 'dst' is not necessary.

3) $MPTsrc = \frac{NEI}{PAsrc}$

4) $MPTdst = MPTsrc \times \frac{PAsrc}{PAdst} = \frac{NEI}{PAsrc} \times \frac{PAsrc}{PAdst} = \frac{NEI}{PAdst}$

MPT = Method Process Time
NEI = Number of Executed Instructions
PA = Processor Availability

5) $MTTsrc = \frac{SSP}{NAsrc}$

6) $MTTdst = MTTsrc \times \frac{NAsrc}{NAdst} = \frac{SSP}{NAsrc} \times \frac{NAsrc}{NAdst} = \frac{SSP}{NAdst}$

MTT = Method Transfer Time
SSP = Size of Serialized Parameters
NA = Network Availability

7) $PAsrc = HPAsrc + RPUsrc$
8) $PAdst = HPAdst + RPUdst$

9) $HPAsrc = HPCsrc - HPUsrc$
10) $HPAdst = HPCdst - HPUdst$

PA = Processor Availability
HPA = Host Processor Availability
RPU = Runtime Processor Usage
HPC = Host Processor Capacity
HPU = Host Processor Usage

11) $NAsrc = (HNAsrc < HNAop)\ ?\ HNAsrc : HNAop$
12) $NAdst = (HNAdst < HNAop)\ ?\ HNAdst : HNAop$

13) $HNAop = HNCop - HNUop + RNUop$
14) $HNAsrc = HNCsrc - HNUsrc + RNUsrc$
15) $HNAdst = HNCdst - HNUdst + RNUdst$

NA = Network Availability
HNA = Host Network Availability
RNU = Runtime Network Usage
HNC = Host Network Capacity
HNU = Host Network Usage

- The tag 'op' refers to the machine where the opposite object (i.e. caller or callee of a method) is located.

Figure 3-8. Metrics Definitions and Relationships

In order to compute the data transfer time of a method (i.e. MTT), this work uses the Size of Serialised Parameters (SSP) metric which refers to the total size of the serialised/marshalled parameters (including return values) that are transferred between the com-

municating remote methods. As is the case with NEI, SSP was introduced in [146] as a supporting metric. However, unlike NEI which needs to be derived from other metrics, it is straightforward to collect SSP through direct measurement as described in section 4.2.1.3.

Metrics		Description	Unit
Software	Number of Executed Instructions (NEI)	Number of Java instructions of a certain execution	instructions
	Size of Serialised Parameters (SSP)	Size of serialised parameters and return values	bytes
	Processor Usage Time (PUT)	The time that a thread/process spends in using the processor of a machine	ms
	Size of Serialised Object (SSO)	The size of a serialised object	bytes
Resource	Host Processor Capacity (HPC)	The processor capacity of a host	instructions/ms
	Host Network Capacity (HNC)	The network capacity of a host	bytes/ms
	Host Processor Availability (HPA)	The CPU (cycles) availability of a host. This is not to be confused with PA which refers to the availability from the point of view of the application.	instructions/ms
	Host Network Availability (HNA)	The unused network bandwidth of a host. This is not to be confused with NA which refers to the availability from the point of view of the application.	bytes/ms
	Host Processor Usage (HPU)	The processor load of a host	instructions/ms
	Host Network Usage (HNU)	The network usage of a host	bytes/ms
	Runtime Processor Usage (RPU)	The CPU load of a runtime (i.e. the application running in the runtime)	instructions/ms
	Runtime Network Usage (RNU)	The network usage of a runtime (i.e. the application running in the runtime)	bytes/ms
Refer to Table 3-2 for a list of the *core* metrics			

Table 3-3. Additional Metrics for the Complete Version of the Proposed Formula

Another improvement that is included in the proposed algorithm is the use of IF (Invocation Frequency) in place of NI (Number of Invocations), to show the significance of a method. As mentioned in section 3.1.2, NI has several limitations which originate from the fact that it does not reflect the temporal aspect of the invocations. IF addresses this issue by dividing the measured number of invocations with the duration of the measurement, which effectively results in the number of invocations per time unit (i.e. millisecond). One advantage of the division is that the resulting calculation is not biased towards objects that have longer life spans. Moreover, this approach provides the possibility of splitting the IF collec-

tion into smaller interval windows (e.g. every N seconds) rather than for the whole duration of the application execution. This flexibility allows decision making to place more weight/importance on newer IF values by applying certain time-series formulas such as the exponentially weighted moving average, which will be presented in section 4.1.4 on *metrics representation*.

* See Table 3-2 and
Table 3-3 for information on the required metrics.

$$(1.1)\ ORUIsrc = \sum_{i=1}^{Methods}\left(\sum_{j=1}^{RCallees\_src}\left(IF_{ij}\times MTTsrc_{ij}\right)+\sum_{j=1}^{RCallers\_src}\left(IF_{ij}\times MTTsrc_{ij}\right)+\left(IF_i\times MPTsrc_i\right)\right)$$

$$= \sum_{i=1}^{Methods}\left(\sum_{j=1}^{RCallees\_src}\left(IF_{ij}\times\frac{SSP_{ij}}{NAsrc}\right)+\sum_{j=1}^{RCallers\_src}\left(IF_{ij}\times\frac{SSP_{ij}}{NAsrc}\right)+\left(IF_i\times\frac{NEI_i}{PAsrc}\right)\right)$$

$$= \sum_{i=1}^{Methods}\left(\sum_{j=1}^{RCallees\_src}\left(IF_{ij}\times\frac{SSP_{ij}}{NAsrc}\right)+\sum_{j=1}^{RCallers\_src}\left(IF_{ij}\times\frac{SSP_{ij}}{NAsrc}\right)+\left(IF_i\times\frac{PUTsrc_i\times HPCsrc}{HPCsrc-HPUsrc+RPUsrc}\right)\right)$$

```
Where:
NAsrc = (HNAsrc < HNAop) ? HNAsrc : HNAop
```

$$(1.2)\ ORUIdst = \sum_{i=1}^{Methods}\left(\sum_{j=1}^{RCallees\_dst}\left(IF_{ij}\times MTTdst_{ij}\right)+\sum_{j=1}^{RCallers\_dst}\left(IF_{ij}\times MTTdst_{ij}\right)+\left(IF_i\times MPTdst_i\right)\right)$$

$$= \sum_{i=1}^{Methods}\left(\sum_{j=1}^{RCallees\_dst}\left(IF_{ij}\times\frac{SSP_{ij}}{NAdst}\right)+\sum_{j=1}^{RCallers\_dst}\left(IF_{ij}\times\frac{SSP_{ij}}{NAdst}\right)+\left(IF_i\times\frac{NEI_i}{PAdst}\right)\right)$$

$$= \sum_{i=1}^{Methods}\left(\sum_{j=1}^{RCallees\_dst}\left(IF_{ij}\times\frac{SSP_{ij}}{NAdst}\right)+\sum_{j=1}^{RCallers\_dst}\left(IF_{ij}\times\frac{SSP_{ij}}{NAdst}\right)+\left(IF_i\times\frac{PUTsrc_i\times HPCsrc}{HPCdst-HPUdst+RPUdst}\right)\right)$$

```
Where:
NAdst = (HNAdst < HNAop) ? HNAdst : HNAop
```

$$(1.3)\ MC = \log MIT$$

$$= \log\frac{SSO}{NA}$$

```
Where:
NA = (HNAsrc < HNAdst) ? HNAsrc : HNAdst
```

Figure 3-9. Final Form of the Proposed Formula

Multiplying IF with MPT (Method Process Time) produces a value that partially contributes to MRUI (Method Resource Usage Intensity). More specifically, it results in the MRUI

portion defining the intensity of *processor-bound* operations. Similarly, the portion for *network-bound* operations is obtained by multiplying IF with MTT (Method Transfer Time).

The calculation of MRUI has to be done at the finest possible granularity in order to achieve maximum accuracy. In the case of processor-bound operations, the calculation should involve only those IF and NEI (which is used to derive MPT) values that are measured for a *specific method* since different methods get called at different rates/frequencies and execute different sets of instructions. On the other hand, the calculation for network-bound operations, should involve only those IF and SSP (which is used to derive MTT) values that are measured for the communication between the method and a *specific caller/callee*. This is so that the co-locality of the method owner (i.e. the adapted object) and the caller/callee can be determined even after the co-locality changes, e.g. due to adaptation/migration. As this requires more detailed information to be recorded and maintained, the collection of the relevant metrics involves higher complexity as discussed in section 4.1.1.2, which addresses the collection of the metrics related to object interactions.

In theory, due to the inverse relationships between IF and MPT/MTT, in sequential applications, ORUI (which is the aggregate of MRUIs) should never exceed the value of 1. For instance, when the IF of a method is high (e.g. 5 invocations per second), MPT/MTT will be low (e.g. at most 0.2 seconds). On the other hand, when the total of MPT and MTT is high (e.g. 10 seconds), IF will only be at most 0.1 invocation per second. However, in practice, this assumption does not always hold as there are two reasons that could cause the value of ORUI to be larger than 1. The first reason is the inaccuracies that are present in the metrics collection process which includes the inaccuracy of the relevant measurement (section 4.3.1) as well as the representation of metrics (section 4.1.4). Another situation where ORUI could exceed 1 is when calculating the ORUI for the destination node (as presented in section 3.2.2.2) in which the resource availability of the destination node is much higher than the source node.

3.2.2.2 Predicting ORUI for the Destination Node

In order to compute ORUI/MRUI for the destination node (i.e. after adaptation/migration), the calculation presented in section 3.2.2.1 should be extended to include *resource metrics* such as processor- or network-related metrics, in addition to the *software metrics* presented in the section (i.e. section 3.2.2.1). This is done in a similar fashion to the original algorithm whereby certain metrics (e.g. MPT) are calculated based on the ratio between the resource availability (e.g. processor availability) of the source and the destination machines for the purpose of estimating the values of the metrics in the destination node. In the case of the proposed algorithm, MPTsrc and MTTsrc (i.e. for the source node) are computed against the ra-

tio of the processor and network availability respectively, in order to obtain MPTdst and MTTdst (i.e. for the destination node), as can be seen from Figure 3-8.

The computation of MPT for the destination node is straightforward since calculating Processor Availability (PA) only requires processor-related metrics/information regarding a specific host (e.g. source or destination host). On the other hand, calculating MTT is more involved since obtaining Network Availability (NA) requires the network-related metrics of the two hosts involved in the remote communication. In particular, the HNA (Host Network Availability) of the two hosts need to be compared, the lower of which will be used to represent the network bandwidth available for the remote communication since that will be the greatest bottleneck of the communication.

It is acknowledged that such an approximation could be inaccurate due to reasons such as the existence of intermediary network devices that bridge the communication between the two machines. Another limitation of the approximation of NA is that it considers only bandwidth, but ignores network latency which also contributes to the overhead of remote communications. However since addressing the mentioned limitations is non-trivial and because the aim of this thesis is to enable live/automated adaptation and demonstrate its usefulness, this limitation is left to future work.

Nevertheless, the lack of latency information is somewhat compensated by the fact that additional information (e.g. metrics propagated from proxies to mobile objects as explained in section 4.2.1.3) needs to be passed during method invocation, which means that remote calls always incur non-zero network overhead even though no parameter is passed by the application itself. Another consequence is that objects exhibiting high frequency of communication are considered to be of higher significance (i.e. bigger impact on application performance) than objects which communicate large data. In particular, an object with X IF (Invocation Frequency) and Y SSP (Size of Serialised Parameters) from an object O is considered to have higher interaction intensity than an object with Y IF and X SSP from O. Note that the mentioned SSP refers to the size of the parameters (and the return value) of the original non-adaptive application, whereas the actual SSP used by in decision making is the collective SSP which includes the additional transmitted information.

In terms of defining/calculating the resource availability of a machine, the original definition used in section 3.1.2 is extended so that resource availability refers to the *entire* resource that is available to the application/object rather than just the *additional* resource, which as explained in section 3.1.2.2, could increase occurrences of unnecessary migration. As shown in Figure 3-8 and Figure 3-9, the modification involves adding the resource that the application process is currently using, e.g. Runtime Processor Usage (RPU) or Runtime Network Usage (RNU), to the calculation of resource availability, in order to compensate for the resources that the application/runtime has been using (i.e. *internal load*) which as a result, low-

ers the resource availability of the host/node. As illustrated in Figure 3-10, such addition allows more accurate calculation of resource availability in comparison to the calculation in the original algorithm (which was described in section 3.1.2.2).

The use of runtime-related metrics (e.g. RPU, RNU) to reflect the internal load of a particular application is sufficient for the scope of this work for two reasons. Firstly, this work assumes that each runtime can only manage a single rather than multiple applications, because the latter is not supported in most mobile object frameworks reviewed in 2.5, with the exception of the MobJeX framework [147]. Furthermore, this work focuses on sequential applications, in which (as argued in section 3.1.2.2), an object can execute (i.e. using the processor and network) with minimal contention from other threads, and thereby implying that the majority of resources that are available to the application, is also available to the object.

On the other hand, this expectation is not applicable for parallel applications wherein there are often other threads which are consuming the resources in parallel with the execution of an object, thus rendering the calculated resource availability inaccurate (i.e. higher than the actual availability since resources are shared by multiple threads). Addressing this issue requires the complex analysis of the relationships between individual threads and objects, similar to the work done in [87] and as such this issue is left for investigation in future work.

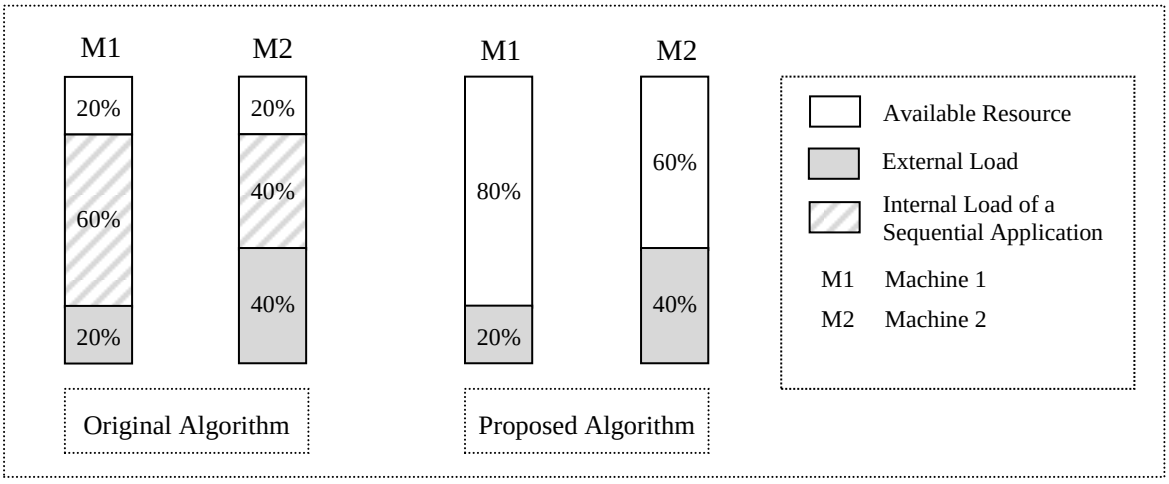

Figure 3-10. Resource Availability in Original versus Proposed Algorithm

Regardless of whether the application is sequential or parallel, the use of runtime resource utilisation metrics leaves a possibility for inaccuracy since the measured RPU and RNU metrics also include the utilisation of the containing middleware/framework due to the middleware functionality executing in the same runtime as the application. This means that the calculated resource availability is slightly higher than the actual amount available to the application but is generally tolerable as long as the middleware does not perform resource intensive operations that are *independent* of the adapted object. On the other hand, middleware operations *related* to the adapted object such as the collection of software metrics (e.g. IF, SSP), do not contribute to the inaccuracy since the resource consumption will follow wherever the ob-

ject migrates thus is considered as a part of the object. Using thread-level resource utilisation metrics should provide more accurate information, however for simplicity, runtime-level metrics are used in this work because of the implementation/technical issues described in section 4.3.1.

3.2.2.3 Calculating Migration Cost

As shown in Figure 3-9, the cost of migrating an object is calculated using the Size of Serialised Object (SSO) metric, which refers to the size of the serialised form of the object. For a similar reason to the selection of SSP (Size of Serialised Parameters), the serialised size is preferred to the in-memory size since it more accurately reflects the number of bytes that will be transferred to the destination node during the migration of the object. In the calculation, the SSO of the adapted object is firstly divided with NA (Network Availability) in order to obtain the estimated time for migrating the object, i.e. MIT (Migrate Instance Time).

The calculated MIT is then provided as an input to a logarithmic function to produce the migration cost. A logarithmic function is used instead of more "sensitive" functions such as linear functions since the latter are more susceptible to having a *strict upper limit* whereby objects with sizes larger than the limit will never migrate unless the network availability is comparably high, but in reality the associated one-off cost may be acceptable considering that the gained benefit (i.e. performance improvement) is cumulative (over the remaining period of execution). In this solution, the log base should be configurable so that it can be set to a value that is most appropriate for the application (i.e. according to the characteristics of the application and its constituent objects). To ease such configuration, the log base can be automatically calculated based on the *expected cost* for a specific *migration time* specified by the application deployer.

A practical example of such a configuration is that if the average performance improvement (i.e. the difference between *ORUIsrc* and *ORUIdst*) gained from adaptation is identified to be roughly 0.1, the deployer could specify that a *migration time* of 2 seconds should translate to the *migration cost* of 0.1. Such a configuration implies that a 2-second migration is considered to be the acceptable upper limit, which means that a longer estimated time (i.e. > 2 seconds) would not trigger a migration except in special cases where the destination node could offer an improvement of more than the average of 0.1. Note that for the purposes of the discussion, this is a simplified case in which, the adaptation score threshold addressed in section 3.2.2.4 is not considered (i.e. assumed to be non-existent or zero).

The appropriate log base for the scenario described above is 2^{10}, which is derived from the two input properties specified by the application deployer, i.e. the *acceptable migration time limit* and the *expected migration cost* for that time limit. The formula is as follows:

$$LogBase = MigrationTime^{(\frac{1}{MigrationCost})}$$

Note that in practice, migrating a mobile object (e.g. a Java object) might require the classes of the object to be transferred and loaded at the destination node, which adds to the cost of migrating the object. However, obtaining MCT is difficult since it is hard to predict whether the required classes (e.g. Java classes) might or might not need to be migrated since this depends on whether the compatible classes are already in the classpath of the destination runtime. Due to this complexity, this implementation issue is left to future work.

3.2.2.4 Determining Score Threshold

The following discussion concerns the configuration of the *adaptation score threshold*, which serves to determine whether a certain object migration should take place depending on the relative difference between the calculated/expected performance improvement and the specified threshold as explained in section 3.1. The specification of the *score threshold* in the proposed solution is more intuitive compared to the original solution, which requires an arbitrary trial-and-error configuration as explained in the next paragraph. On the other hand, even though determining the appropriate thresholds in the proposed solution might require several iterations of fine-tuning/reconfiguration, logical reasoning can be applied when specifying the initial score threshold.

The main reason for this advantage is the better comprehensibility of ORUI (i.e. used in the proposed solution) compared to ORT (i.e. the original solution), which represents the response time of an adapted object accumulated during an arbitrary/indefinite period of time. On the other hand, ORUI is clearly defined as the degree of resource utilisation of an object per time unit (i.e. millisecond), which as explained in section 3.2.2.1, has a theoretical upper limit of 1 which refers to a 100% utilisation of resources during the whole sampling period (i.e. the period in which the relevant metrics are collected).

Common factors that should be considered in configuring the score threshold include the average usage intensity of objects and the lowest acceptable improvement. For example, in a simplified scenario where the deployed application consists of 5 objects of equal characteristics (i.e. high execution intensity) and execution probability, the average usage intensity of an object can be assumed to be roughly equivalent to 0.2. In the case where the lowest acceptable improvement is set to be 20% (i.e. objects should migrate only if at least 20% performance improvement can be gained), the score threshold would be specified as 20% of 0.2 which is 0.04.

Having presented an algorithm for local adaptation via application partitioning, the next chapter will address the automatic collection and management of information that is required by the proposed algorithm.

Chapter 4. Metrics Management

This chapter presents a context management solution for the primary purpose of automating the local-adaptation solution proposed in section 3.2. Note that although automating such adaptation solution might also require a robust fault-tolerance mechanism, the discussion of such a topic is beyond the scope of this thesis.

Section 4.1 presents a generic solution for addressing the efficiency and accuracy of the management of context (in the form of metrics). Although primarily targeting the proposed adaptation solution, this solution can be extended to facilitate other adaptation solutions, such as the original solution described in section 3.1 or a global-adaptation solution, and therefore where possible, its flexibility and generality will also be discussed.

Section 4.2 extends the solution presented in section 4.1 to specifically facilitate the management of metrics required by the proposed adaptation solution (presented in section 3.2). Finally, section 4.3 explains the implementation of the presented metrics management solution with emphasis on how they were implemented in Java and in a mobile object framework called MobJeX. The evaluation of the metrics management solution/implementation in terms of the incurred resource utilisation overheads (e.g. memory usage) and its impacts on the performance (i.e. response time) of the supported application, is presented in Chapter 5 (along with the evaluation of the proposed adaptation algorithm).

4.1 General Overview

Challenges involved in the management of metrics-related operations include those for: 1) collecting metrics through measurement (addressed in section 4.1.1), 2) delivering collected metrics to adaptation engines (section 4.1.2), 3) controlling individual management tasks using customisable criteria (section 4.1.3), and 4) representing metrics for the purpose of capturing temporal information (section 4.1.4).

The adopted solution for addressing the aforementioned challenges, is based on the object-oriented context modelling approach (introduced in section 2.4) as illustrated in Figure 4-1. Consequently, objects are used to represent relevant information encapsulated in the form of *model entities*, *metric containers*, and *metrics*. Model entities serve to model all middleware/application components in the system (which as mentioned in section 2.5.1, includes host managers, runtimes, and mobile objects) and their hierarchical relationships. Metric containers are used to manage the collection of metrics belonging to specific components. Lastly, a metric object encapsulates data and operations related to the collection of a particular metric (e.g. Invocation Frequency).

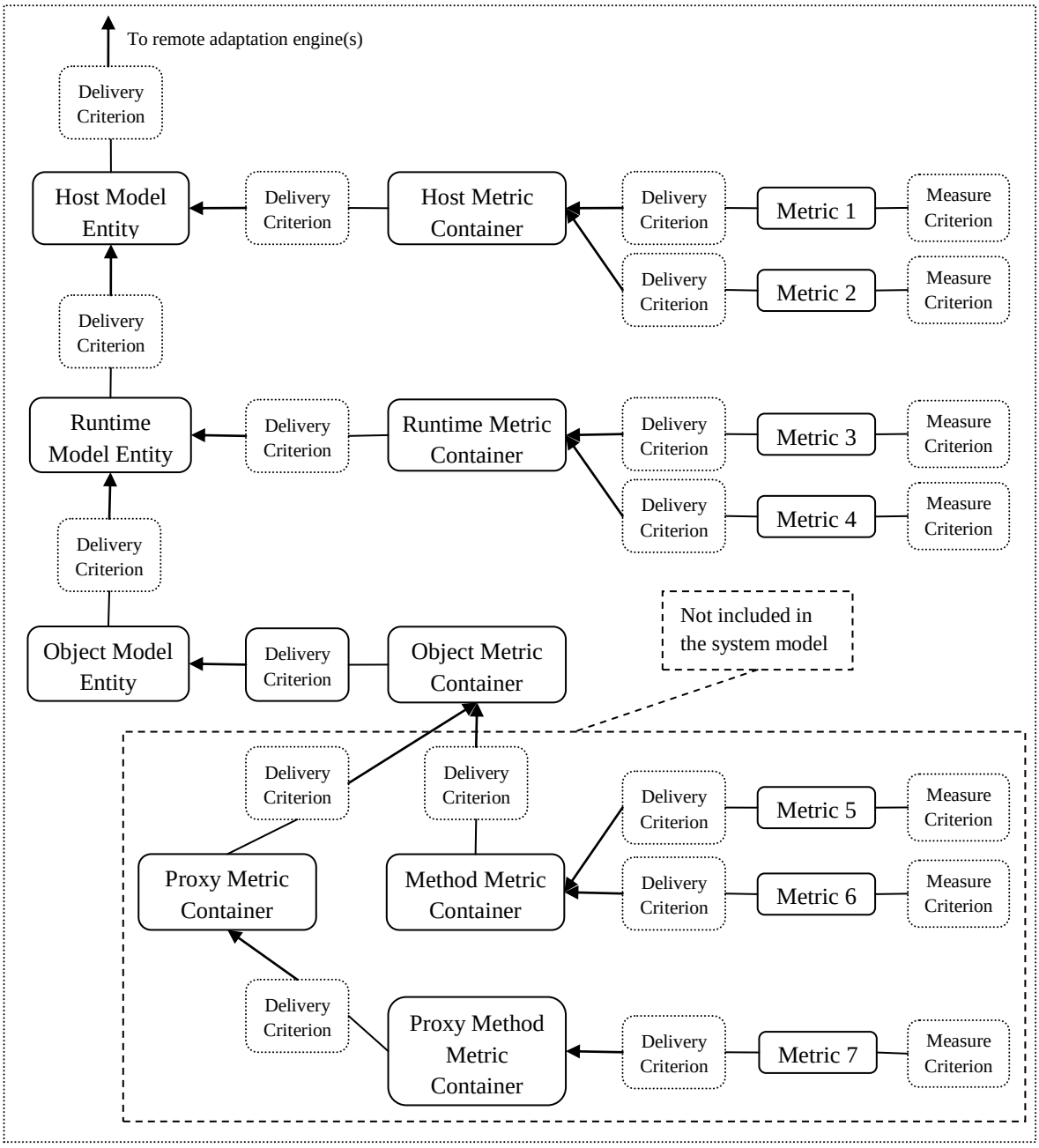

Figure 4-1. Generic Metrics Collection Process

As can be seen in Figure 4-1, a host model entity, which represents a specific host manager, relies on a host metric container to manage the collection of host-related metrics (e.g. Host Processor Usage). Similarly, a runtime model entity represents a particular runtime and manages its metrics through a runtime metric container. The parent-child relationship between the host manager and its runtime(s) is also captured in the relevant model entities. Each mobile object in the application should also have a corresponding object model entity. Object model entities are not required for other types of objects such as stationary remote objects

and non-remote objects, which as mentioned in section 2.5 are not adaptable (i.e. cannot migrate) and therefore do not need to collect metrics.

The aforementioned model objects (i.e. model entities, metrics and their containers) as well as their interconnections are maintained (as a *system model*) in an adaptation engine to facilitate the analysis of system state for the purpose of making adaptation decisions. Due to the complex relationships represented by the system model, the extraction of metrics from the model is achieved using the *visitor design pattern* [60], in which a visitor object is used to assist the traversal of the model, starting from the root of the model (i.e. service model entity) down to the individual leaves (i.e. mobject model entity). The extracted metrics are then used for adaptation processing purposes such as calculating the performance scores of mobile objects (as discussed in Chapter 3).

Depending on the chosen adaptation scheme (e.g. local versus global), multiple adaptation engines might exist, in which case multiple system models are managed across different adaptation engines. Furthermore, each component in the system may maintain its own model, which is a subset of the entire system model. For example, a runtime may maintain a runtime model entity, whereas a host manager may maintain a host model entity containing the model entities of its runtimes.

Note that even though proxies of mobile objects (introduced in section 2.5.1) also play a role in the collection of metrics as will be discussed in section 4.2.1.3, they are not included in the system model because of the difficulty/complexity involved in maintaining the relevant model entities due to the following reasons. Firstly, since a proxy is essentially a more sophisticated form of regular object references, as is the case with object references, there can be multiple references/proxies to the same mobile object. Furthermore, references/proxies can be volatile in that they can be created and destroyed at any time (e.g. due to code scoping).

Despite the exclusion of proxy model entities, proxy metric containers, which are responsible for managing proxy-related metrics collection tasks, are still considered in the metrics collection process. Unlike other metric containers (i.e. those belonging to mobile objects, runtimes, and host managers), proxy metric containers are not part of the model. However, the inclusion of a proxy metric container in Figure 4-1 serves mainly to capture certain delivery relationships which as will be discussed in section 4.1.2, define how metrics are delivered from the originating/measuring component to the adaptation engine. Figure 4-1 also shows the use of *criteria* at different places, which as discussed further in section 4.1.3, serve to control certain metric-related operations, usually for efficiency purposes.

4.1.1 Metrics Collection

Section 4.1.1.1 describes the adopted metrics collection approaches, the flexibility of which is demonstrated in section 4.2.1 through the discussion of the collection of various metrics required by the proposed adaptation algorithm (presented in section 3.2). On the other hand, section 4.1.1.2 discusses specific approaches used to address the collection of metrics related to object interaction, such as Size of Serialised Parameters (SSP) and Invocation Frequency (IF).

4.1.1.1 Common Approaches

Since this work adopts the object-oriented context modelling approach, each metric is encapsulated in a specific object/class which is responsible for performing tasks related to the collection of the metric. For instance, the `RMUMetric` class/object is responsible for acquiring memory usage information of a runtime (RMU) and storing the obtained information (in the form of integer-typed *measures*). Note that although RMU is not used in the proposed adaptation score calculation formula, it is used prior to score calculation to determine whether a target machine has sufficient memory to receive a particular mobile object, as described in section 3.2. The encapsulation of metric-specific responsibilities allows different metrics to apply different management policies such as metrics representation or collection/delivery criteria, as discussed in section 4.1.4 and 4.1.3. Furthermore, class inheritance allows metrics of similar types such as host-resource metrics, which include HPU (Host Processor Usage), HNU (Host Network Usage), etc., to be grouped and treated in the same way (i.e. polymorphically).

Considerations involved in metrics collection include determining the location/component where the relevant measurement takes place as well as when the measurement should be initiated. In terms of measurement location, it is most appropriate to collect host/node-related metrics such as HPU and HNU, in the host manager. Similarly, runtime metrics, such as RMU, should be collected in the relevant runtime as discussed further in section 4.2.1. Consequently, as opposed to the illustration shown in Figure 4-1 (of section 4.1), which is presented for simplicity, metrics collection does not occur in a single model, but is rather executed in separate models maintained by individual components.

In terms of its initiation, measurement can be triggered using timers recurring at a regular interval. Such an approach is hereinafter referred to as *time-based* measurement initiation, whereas the relevant collection/measurement is called *timer-initiated* collection/measurement. This is in contrast to *application-based* initiation, in which case, the measurement is initiated by the application (e.g. through method invocations) and hence is also known as *application-initiated* measurement. A hybrid of the two initiation approaches

(i.e. time-based and application-based) is also required for certain metrics (e.g. Invocation Frequency) due to the specific reasons explained in section 4.2.1.

An important characteristic of application-initiated measurement is that its occurrences are heavily dependent on the specific behaviour of the application. For instance, a frequently invoked method would lead to high frequency of measurement which could introduce significant performance overhead. Furthermore, unlike timer-initiated measurement which is handled by a separate timer thread, application-initiated measurement executes on application threads, thereby affecting the performance of the adaptive application. Consequently, as illustrated in Figure 4-1 (of section 4.1), criteria are placed in measuring components (e.g. mobile objects) to control the frequency of the relevant measurement, thereby reducing the associated overheads, as discussed in more detail in section 4.1.3.

4.1.1.2 Approaches for Collecting Interaction Metrics

Interaction metrics, which refer to metrics related to object interaction or method invocation (e.g. Invocation Frequency), should be associated with the caller/callee involved in a particular method invocation, which as mentioned in section 3.2.1, serves to distinguish between *local* and *remote* calls. Since interaction metrics should be collected for both *incoming* and *outgoing* calls, in the scenario where a mobile object invokes a method of another mobile object, the metrics will be collected twice, i.e. once for each mobile object.

Because the measurement for *incoming* calls can be done in the callee object/method itself, the required instrumentation task (i.e. inserting the measurement code into the invoked method) is straight forward. On the other hand, the instrumentation for *outgoing* calls, which requires modification of the caller code, involves higher complexity since such code may belong to an external library rather than the application itself. The instrumentation is further complicated by the fact that method invocations in object-oriented languages such as Java, are polymorphic, thus requiring the instrumented code to be complemented with runtime checking. Furthermore, modifying/instrumenting certain code such as native code (i.e. written in lower-level languages such as C) is impractical if possible at all. Such a limitation affects the transparency of the application development which as mentioned in section 2.6, is also a concern of this thesis due to its impact on the required development effort/cost. The general solution to this type of issue is to bridge the communication between the caller/callee objects using the *proxy design pattern* [60], which fits the role of proxies in mobile object frameworks as discussed in section 2.6.

Since mobile objects are accessed via proxies, the measurement code for outgoing calls can simply be inserted into the proxies, more specifically, before/after the proxy forwards calls to the corresponding mobile object. However, this approach introduces a new problem whereby the collected outgoing-call metrics are (by default) stored in the proxy instead of the

caller object, which is the object associated with the collected metrics. This is especially problematic since as mentioned in section 4.1, proxies are volatile by nature thereby complicating the delivery of metrics to the adaptation engine.

A simple solution involves returning the metrics from the proxy back to the caller, however this affects development transparency since it requires changing the return type of the proxy method, making it incompatible with the original method. Instead, this thesis proposes a solution based upon a registry approach, which has traditionally been used to provide contact points (known as registries) for clients to acquire particular objects based on their associated names. In this work however, a registry is used as the main contact point for a callee object to obtain information regarding its caller.

In doing so, the registry traces the flow of application execution by maintaining a list of all mobile objects that are involved in the execution. The list is managed as a stack reflecting the current execution flow, thereby allowing the caller of a specific object to be discovered by *peeking* at the stack. As shown in Figure 4-2, using this solution, a proxy can pass the outgoing interaction metrics collected in the proxy to the rightful owner of the metrics, i.e. the caller object. In the case where the stack is empty, it is assumed that none of the preceding objects in the call chain (i.e. the caller and its callers) is a mobile object since only mobile objects get pushed into the stack. Other objects (e.g. stationary objects) are not pushed since they are not adaptive, and thus do not need to collect metrics.

It is important to note that a separate stack needs to be maintained for each executing thread since each thread defines a different flow of execution. Such a safe guard is required even though the focus of this thesis is on sequential applications (as opposed to parallel applications), since as mentioned in section 2.2, a sequential application might consists of multiple executing threads.

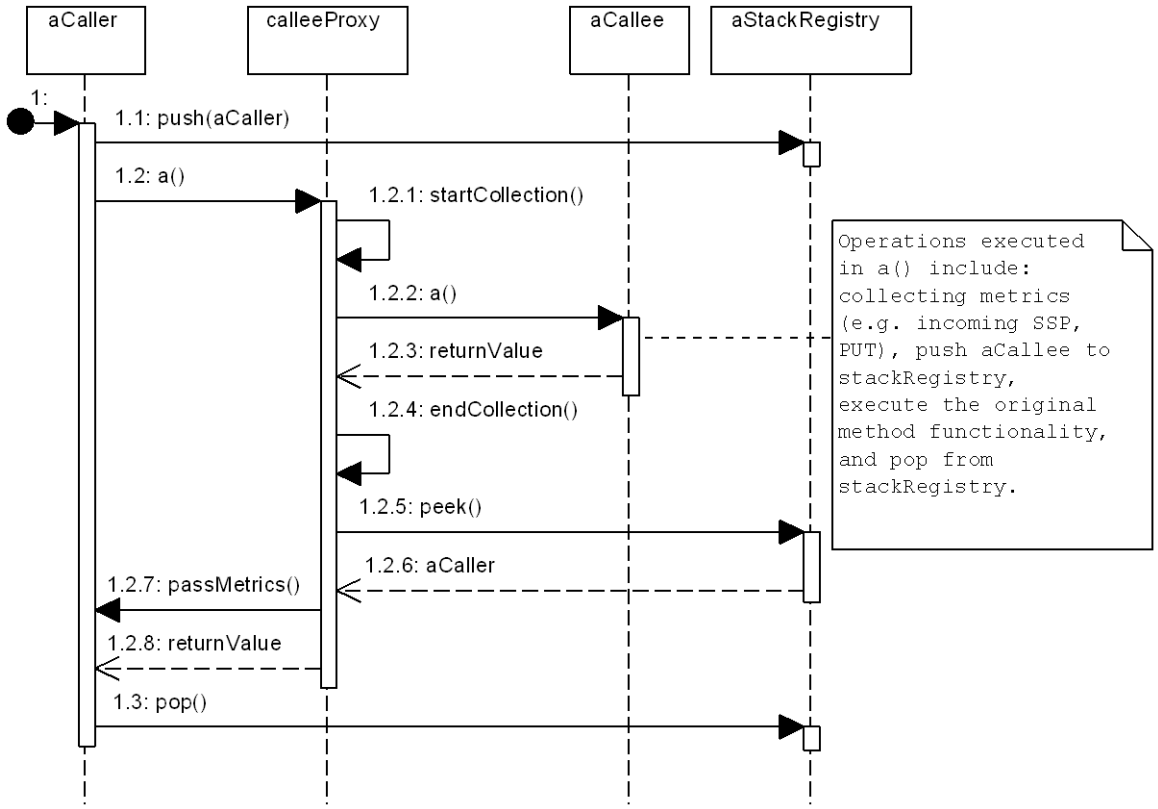

Figure 4-2. Using Stack-based Registry for Collecting Interaction Metrics

In addition to passing information to the caller, the registry has another important role in the collection of interaction metrics (i.e. SSP and IF), which as previously mentioned, need to be associated with the relevant caller/callee. More specifically, the registry facilitates the collection of metrics for *incoming* interactions because the caller object needs to be identified. As previously mentioned, recording the caller object allows the adaptation engine to determine whether the relevant call is local or remote depending on the current location of the caller.

Since each registry is used to maintain information about the execution of a particular thread, its information is not distributed across machines. In particular, a mobile object that is called remotely will not be able to access the caller through its registry, since the registry of the caller is located on another machine. Consequently, certain information required for identifying the caller (e.g. a unique number) needs to be passed to the callee during the invocation of the relevant method. Note that, since this can be done without modifying the method signature of the proxy, existing code will not be affected, thereby providing development transparency.

In some cases, the proxy could fail to retrieve the caller information from the registry stack due to it being empty because none of the preceding objects is mobile. In such a situation, the information regarding the current runtime/process (of the caller) would get used in-

stead. Using this information, the co-locality of the call (i.e. local versus remote) can still be accurately determined during adaptation decision making since stationary objects will always execute in the same runtime. In the case of *outgoing* interactions, the registry is not needed since the relevant callee can be determined via its proxy since a proxy should have information about the object, to which it is referring.

This registry-based solution is used for the collection of interaction metrics (e.g. SSP and IF) as discussed in detail in section 4.2.1.3.

4.1.2 Metrics Delivery

The delivery of metrics from one to component to another is illustrated in Figure 4-1 (of section 4.1) as a unidirectional relationship/arrow connecting model entities, metric containers, and metric objects. Note that even though the diagram (Figure 4-1) only shows the delivery relationships between model entities (e.g. host model entity), in practice, metrics delivery might also involve interaction between the represented components (e.g. host manager). As shown in the diagram, metrics are collected in individual components (including proxies) and then delivered to the parent components, using either a *push* or a *pull* approach (introduced in section 2.4.3).

The push strategy involves a chain of propagations to convey metrics that are measured in a particular component to the immediate parent component. Since such a strategy enables *delivery upon update*, whereby metrics are delivered when there are updates or changes, it plays an important role in adaptation initiation by allowing decision making to be triggered when the newly received metrics satisfy certain criteria (e.g. upon detection of low resource availability), as discussed further in section 4.1.3. On the other hand, a pull strategy which allows *delivery upon demand*, is particularly useful for reducing unnecessary deliveries (thus the implied performance overhead) as applied in section 4.2.2.

The delivery overhead is of particular concern in *inter-process delivery* in which metrics are delivered to a receiving component running in a different process space to the sender. Unlike *local delivery*, in which metrics are delivered between components running in the same process via reference passing, inter-process delivery involves serialisation and deserialisation of metrics, which introduces considerable performance overhead even though network communication is not involved.

In particular, the cost of delivering metrics from a mobile object to its managing runtime is negligible (since both execute in the same process), but this is not necessarily the case with the delivery from a runtime to the host manager, since in practice, a host/machine can only have one host manager but can run multiple applications/runtimes, thus requiring execution in separate processes. The cost is even higher in *inter-host delivery*, which is a specific form

of inter-process delivery, involving components executing on different machines, thereby requiring network communication. In global adaptation, such a case applies to the delivery of metrics from individual host managers to the central adaptation engine, whereas in local adaptation, this applies to metrics delivery amongst distributed adaptation engines as discussed further in section 4.2.2.

Consequently, in order to reduce the potentially high overhead of metrics delivery, the aforementioned *criteria* approach could also be applied, as described in section 4.1.3. Furthermore, since the overhead introduced by *inter-process* and *inter-host* delivery is particularly affected by the size of the pushed metrics, *fine-grained* delivery is favoured in this work over coarse-grained delivery. In *fine-grained* delivery, metrics are delivered either as individual metric objects (e.g. `RMUMetric`) or as a group of metric objects. On the other hand, in the *coarse-grained* approach, the delivery involves an entire model entity (e.g. host model entity), which includes a container of its metrics (e.g. host metric container) as well as the contained child entities (e.g. the model entities of all the runtimes managed by the service). The main advantage of such an approach is that the containment relationships between the interconnecting model entities (i.e. parent and child entities) as well as all their metrics will get delivered to the adaptation engine as a cohesive bundle, thereby allowing the adaptation engine to directly use the received model entity (including its child entities) rather than having to reconstruct certain information (e.g. relationships between entities).

However, since in practice, the overhead of marshalling/serialising the complicated relationships as well as synchronising the marshalling process (i.e. to prevent inconsistency due to concurrent updates on the model entity) outweigh the provided benefits, fine-grained delivery is favoured in this thesis. In the fine-grained approach, each delivered metric (e.g. runtime processor usage) is accompanied with certain metadata such as a Uniform Resource Identifier (URI), to identify the measured entity (e.g. a particular runtime). This enables the adaptation engine to reconstruct the relationships between system components and their metrics (by allowing correct placement in the model) to facilitate adaptation decision making.

4.1.3 Management Criteria

This work proposes the use of criteria in the form of self-contained *strategy* objects [60] for controlling operations related to the management of metrics (e.g. collection and delivery of metrics). As shown in Figure 4-1 (of section 4.1), criteria are attached to individual metrics to control the collection of the corresponding metrics. Similarly, criteria are also attached to metrics containers and model entities in order to control the delivery of metrics.

The criteria approach allows the application deployer to specify the criterion for a particular operation based on the specific knowledge of the application and the target plat-

form/environment. Although the primary objective is to reduce unnecessary overhead (thus improve efficiency), criteria also enable the encapsulation of other type of functionality such as that for triggering adaptation, as demonstrated in the solution presented in section 4.2.3.

Criteria can be implemented in any form, ranging from a simple filtering criterion aimed at reducing the frequency of a specific operation (for the purpose of reducing the associated overheads), to a more complicated implementation involving aggregation of metrics, similar to the concept of context aggregation used in [112], in which context information is gathered at a specific cluster node prior to its delivery to the parent cluster. In this work, metrics aggregation is used to reduce the overall delivery/communication overhead, as discussed further in section 4.2.2, which presents the criteria designed according to the specific characteristics of the proposed adaptation solution.

4.1.4 Metrics Representation

Adaptation decision making could be improved by providing the adaptation engine access to the temporal information of metrics because it allows the change/update trends of the metrics to be obtained/analysed. For instance, when temporal information is not used but rather the last recorded metric value is used, the adaptation decisions become too *agile* [121] [120], i.e. sensitive to metric changes. This sensitivity causes the application to respond/adapt not only to "real" changes in metrics, but also to sudden and temporary changes, which would result in inaccurate adaptation decisions and thus unnecessary migrations.

The most basic approach to capture the temporal information of a metric is by maintaining a history of the metrics values that are recorded during the application execution. However, this approach introduces significant management overheads, most notably on the memory required for maintaining the history and/or the network bandwidth for delivering it to the adaptation engine. Consequently, this work adopts an approach in which a *representation* is used to efficiently capture the temporal characteristics of the recorded values of a metric (i.e. measures). Simple approaches for building/maintaining the representation include summing or averaging up all the recorded measures to date.

The effect of using such a representation is that the importance/weight of the newer values becomes less/lower as time elapses, due to the accumulation of old values. This makes the application less *agile*, i.e. less sensitive to metric changes, which is the opposite of the adaptation behaviour exhibited when no temporality is used at all, i.e. using last value only. Consequently, an exponentially weighted moving average (EWMA) function was chosen to overcome the mentioned agility-related limitations since it decreases the significance of the older values and as a result places more weight on newer values. EWMA has been used in similar work [87], however the specifics of how it was applied and what effects it had to the

calculation of the relevant metrics were not provided. Consequently, these issues are discussed below.

The overhead of maintaining a metric representation using EWMA is very low, especially compared to maintaining a history of metrics, because the calculation involves only the old average and the latest metric value as shown in the formula in Figure 4-3. The degree to which the weight of the older value decreases is determined using a smoothing factor α, whereby the higher the value of α, the faster the rate of decrease of the older values. The value of α ranges between 0 and 1 and should ideally be configured by the application deployer depending on the nature (e.g. how dynamic) of the application and the execution environment.

$$Avg_t = \alpha \times New + (1 - \alpha) \times Avg_{t-1}$$

```
Where:
Avg = The calculated average value at a specific time period
New = The last collected measure
α   = The chosen smoothing factor
t   = A time period t
```

Figure 4-3. The EWMA Formula

The formula does not explicitly take into account the collection time of the values since the values are generally assumed to be collected at regular time intervals. As such, even though the formula works well with metrics (e.g. Host Processor Usage) collected using a regular interval timer, it is not necessarily suitable for application-initiated measurement (concerning software metrics such as Processor Usage Time and Size of Serialised Parameters), since as mentioned in section 4.1.1, such measurement is triggered upon method invocation which does not necessarily occur at uniform intervals.

The issue could be addressed by dynamically determining α based on the elapsed time between the latest method invocation and the previous invocation, which will effectively adjust the decrease rate of the old average value according to its recentness as shown in Figure 4-4. In the modified formula (equation 1), α is defined as the measurement duration d divided by the maximum acceptable duration m. As an example, if 3600 seconds is chosen as the maximum duration (i.e. $m = 3600$), a measurement that occurs 900 seconds (i.e. $d = 900$) after the last measurement will be calculated using $\alpha = 0.25$. In the case where the elapsed duration exceeds the maximum duration, a maximum α value of 1 is used as shown in equation 2.

$$(1) Avg_t = \frac{d}{m} \times New + \frac{m-d}{m} \times Avg_{t-1}\,, \qquad d < m$$

$$(2) Avg_t = New\,, \qquad d \geq m$$

```
Where:
Avg = The calculated average value at a specific time period
New = The last collected metric value
t   = A time period t
d   = Duration since last measurement
m   = Pre-defined maximum tolerable duration
```

Figure 4-4. The EWMA Formula for Metrics Collected at Irregular Intervals

Although this formula (which targets irregular-interval measurement) has been implemented, a thorough evaluation of the formula is beyond the scope of this work since the aim is to demonstrate the role of metric representations in addressing the aforementioned *agility* issues. Nevertheless, since IF is collected at a uniform interval (as discussed in 4.2.1.3), applying EWMA to IF would capture certain trends in the software metrics (e.g. access patterns). Doing this should sufficiently assist the adaptation decision making since both of the calculation of the *execution intensity* (i.e. MPT) and *interaction intensity* (i.e. MTT), involves IF, thus the aforementioned limitation concerning Processor Usage Time (PUT) and Size of Serialised Parameters (SSP) can be compensated.

Due to code encapsulation being an advantage of object-oriented context modelling, switching to a different representation (in the future) requires minimal maintenance effort. Furthermore, encapsulation also allows the use of different strategies (e.g. simple averaging) and parameters (e.g. α value) for different metrics, which is especially useful in the case where certain strategies are not applicable to the concerned metrics. As an example, the presented EWMA strategy only applies to metrics measured in *ratio* measurement scale (discussed in section 2.4.2), since the calculation involves multiplication and division. Therefore, in the case where a metric is of a less powerful measurement scale (e.g. the *interval* scale), a different representation strategy is required. However, this is not a concern in this thesis since all the metrics required by the adaptation algorithms presented in sections 3.1 and 3.2 are of ratio scale.

4.2 Solution for Local Adaptation

A specialisation of the general metrics management solution presented in section 4.1, is provided in this section to address the management of metrics required by the proposed adaptation algorithm (presented in section 3.2). In particular, section 4.2.1 addresses the collection of the required metrics, whereas section 4.2.2 discusses the optimisation of metrics delivery

based on the specific characteristics of the adopted adaptation solution (e.g. local adaptation). Finally, section 4.2.3 presents a discussion of the adopted metrics criteria such as the *initiation criterion*, which has the role of controlling the triggering of adaptation decision making.

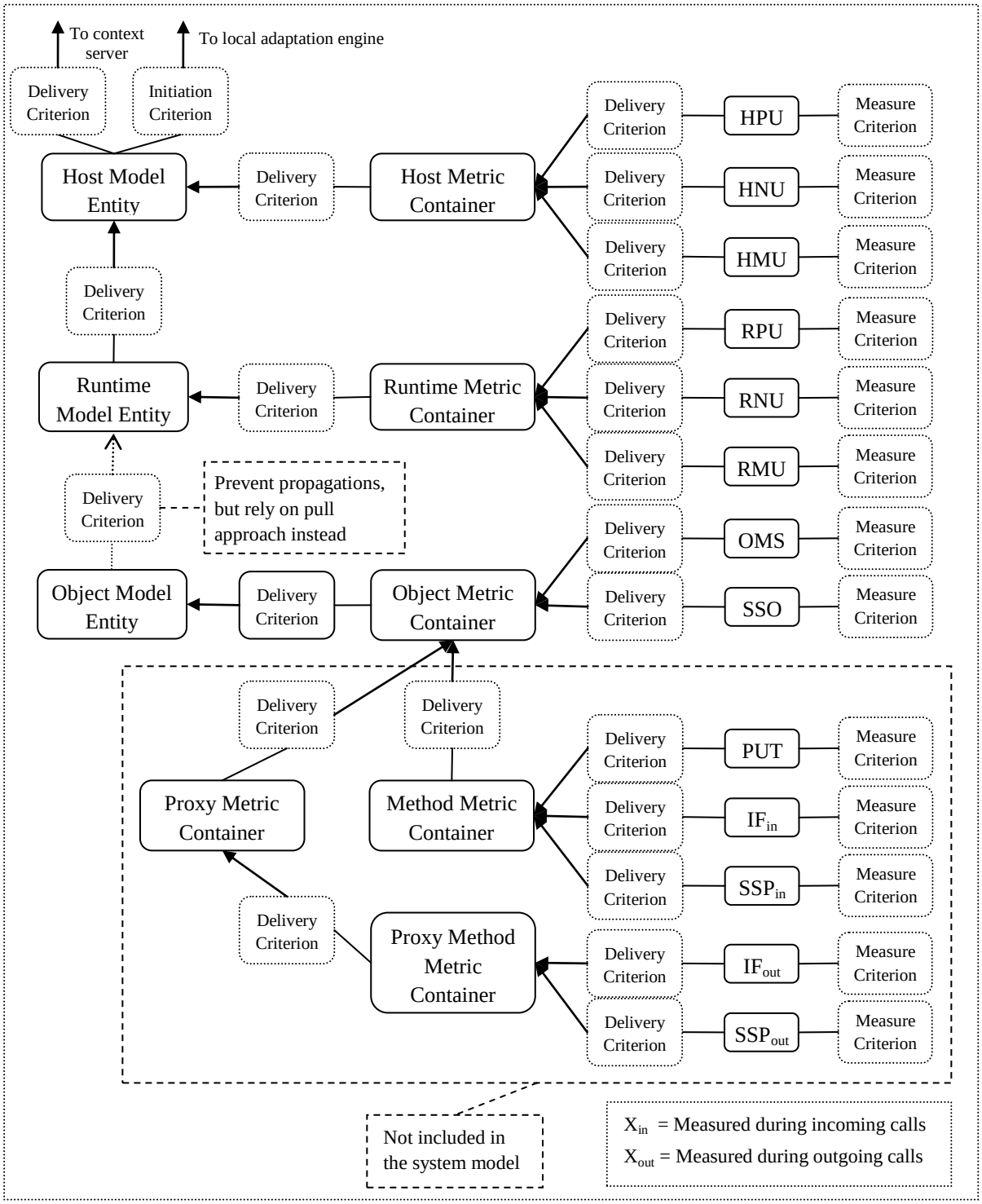

Figure 4-5. Metrics Collection Process for Local Adaptation

Figure 4-5 provides an illustration of the solution, listing all relevant metrics and their relationships with metric containers and model entities. Other aspects shown in Figure 4-5, such as the delivery of metrics from the host model entity to a context server, will be discussed in the relevant section (e.g. 4.2.2).

4.2.1 Metrics Collection

Since the calculation required for obtaining *derived metrics* has been addressed in section 3.2 in sufficient detail, this section is concerned with the collection of *base metrics*, which are acquired through direct measurement. As such, metrics that fall into the *scoring* category, which include ORUI (Object Resource Usage Intensity), MRUI (Method Resource Usage Intensity), and MC (Migration Cost), are excluded from subsequent discussion. Similarly, the discussion also excludes *software* or *resource* metrics obtained through derivation, which as mentioned in section 3.2.2 include MTT (Method Transfer Time), MPT (Method Process Time), NEI (Number of Executed Instructions), PA (Processor Availability), NA (Network Availability), HPA (Host Processor Availability) and HNA (Host Network Availability).

The rest of the metrics, which are collected using direct measurement techniques, will be described in the following subsections, which begin with a discussion of the collection of *resource* metrics (in section 4.2.1.1). The collection of software metrics that are not related to object interaction is discussed in section 4.2.1.2. Finally, the collection of interaction metrics, which uses the registry-based approach described in section 4.1.1.2, is addressed in section 4.2.1.3.

4.2.1.1 Resource Metrics

The collection of resource metrics applies a *time-based* initiation approach (introduced in section 4.1.1.1), in which a recurring timer is used to poll the most relevant computing entities (e.g. OS) for the most current information (i.e. measures). Particularly, the collection of process-related metrics, RMU (Runtime Memory Usage) and RMC (Runtime Memory Capacity), is achieved by retrieving certain information from the relevant runtime. In the case of host-related resource metrics such as HPU (Host Processor Usage), HNU (Host Network Usage), and HNC (Host Network Capacity), the information is simply obtained from the operating system. As is the case with the majority of metrics concerned in this work, HNC is considered as a *dynamic metric* (as opposed to *static metric*), especially in a wireless configuration, which is heavily dependent on factors such as signal strength. As such, its collection should be performed repeatedly though not necessarily as frequently as more transient metrics such as HPU and HNU.

Since there is no straight-forward solution for obtaining HPC (Host Processor Capacity) in the unit required by the proposed adaptation algorithm (i.e. instructions per second), the

measurement requires intervention from the application deployer as will be discussed in detail in section 4.3.1. However, this will not compromise the automation of the adaptation because HPC is generally unchanged, except for high-end servers supporting CPU hot swapping, and thus can be treated as a *static metric*, which is collected once, prior to application execution (i.e. offline).

4.2.1.2 Non-interaction Software Metrics

Similar to the collection of resource metrics, the *time-based initiation* approach is also used for collecting specific software metrics, namely SSO (Size of Serialised Object) and OMS (Object Memory Size), in which, relevant information is acquired from each mobile object at a regular interval. The low level details for acquiring the information (i.e. object sizes) will be discussed in section 4.3.1. These are the only two metrics collected at object level rather than at method level as is the case with the rest of the concerned software metrics, which include PUT (Processor Usage Time), SSP (Size of Serialised Parameters) and IF (Invocation Frequency).

Such metrics are collected at method level mainly because the relevant measurements are initiated by the execution/invocation of a specific method (i.e. *application-initiated* measurement). The PUT of a method execution is acquired by first measuring the current usage at the beginning and at the end of the relevant method, and then subtracting the resulting values. An additional task is required if the executed/measured method (i.e. caller) contains invocation to another mobile object (i.e. callee), in which case, the measured PUT of the callee should be subtracted from the caller, in order to separate the processor usage of the two mobile objects (i.e. caller and callee objects), which may migrate independently to different machines.

Since SSP and IF represent attributes related to object interaction, the collection of these metrics, which introduces additional complexity as described in section 4.1.1.2, is addressed in section 4.2.1.3.

4.2.1.3 Interaction Metrics

Obtaining the SSP of a specific invocation is achieved by measuring the serialisation size of the parameters at the beginning of the method call as well as the return value at the end of the call. Due to the necessity of collecting SSP for both incoming and outgoing calls, the involved operations become redundant, which would significantly affect performance when the measured messages (i.e. parameters and return value) contain large/complex objects because measuring such objects involves complex traversal of the (other) objects that are referenced by the measured object.

Consequently, the redundant measurement operations should be prevented by communicating the measurement results between the proxy and the callee rather than repeating the

measurement of the same data. In this approach, prior to the invocation to the callee, the proxy performs the following tasks: 1) measures the parameter size, 2) stores the result in its outgoing SSP data holder (i.e. `SSPMetric` object), and then 3) passes the result together with the invocation parameters to the callee. The callee then: 1) stores the result in its incoming SSP data holder, 2) executes the invoked method, 3) measures the size of the return value, 4) adds the result to the data holder, and 5) passes the result and the return value back to the proxy. Finally, the proxy adds the received measurement result to the relevant/outgoing SSP data holder. Note that although this approach technically increases network usage due to the transferring of extra information (i.e. SSP value), in practice this should be minimal since the value is a single integer.

The SSP collection could be further optimised by measuring the flat/serialised form of the object where possible, instead of the in-memory view of the object which as previously mentioned requires the traversal of its references. This optimisation takes advantage on the fact that any messages (i.e. parameters plus return value) that are transferred over the network have to be marshalled/serialised anyway. Consequently, acquiring SSP using such an approach is as simple as monitoring the number of bytes that pass through the socket used for transferring the messages. The overhead of such measurement is minimal, but this technique is only applicable for remote communications. This technique however, has not been implemented in this work due to the complexity concerning the delivery of the measurement result from the measuring socket (which is exclusively managed by Java RMI) to the mobile object.

In contrast to SSP, the performance overhead of collecting IF is minimal because it simply involves maintaining a counter (i.e. number of invocations) over a period of time (i.e. sampling period) and then dividing the counter with the sampling period. As such, there is no real incentive for removing the redundant measurements (for incoming and outgoing calls), which would increase the complexity of the solution. Keeping track of the number of invocations is achieved by simply incrementing the relevant counter at each method invocation. Two alternatives for deciding when IF should be calculated (i.e. dividing the invocation counter with the sampling period), were considered.

The first alternative uses a count threshold wherein the calculation is performed when the counter reaches a certain limit, i.e. after a certain number of invocations. In this *invocation-based* approach, specifying a threshold of 1 means that IF will be calculated for every invocation. Since in this approach, IF will not be collected until the relevant method gets invoked, the currently recorded IF values might not be *temporally accurate* in that they no longer accurately represent the most current interaction behaviour. Furthermore, this approach is not suitable for applications containing methods that get executed frequently and rapidly, because this will result in frequent calculation of IF which although not expensive, could potentially

trigger numerous metric deliveries (depending upon how criteria have been set), which might be costly depending on the location of the target component as explained in section 4.1.2.

Consequently, this work applies an alternative *time-based* approach which utilises a timer to trigger the calculation of IF at a fixed interval, thereby implying a constant overhead which is not affected by application behaviour as is the case with the invocation-based approach. Furthermore, this approach produces more temporally accurate metrics since IF will always be collected whether or not the relevant method gets invoked. Note that such a collection approach is a hybrid between the *application-initiated* measurement approach (which is applied for keeping track of the invocation counter) and the *timer-initiated* measurement approach (which is applied for triggering the calculation of IF). In terms of the implementation of such an approach, a scalability issue related to the utilisation of timers will be addressed in section 4.3.1.

4.2.2 Metrics Delivery

Depending on their relevance to the initiation of adaptation, metrics required by the proposed adaptation algorithm, are categorised in two groups, in order to determine how often certain metrics need to be delivered to the local adaptation engine. The first group consists of metrics that are required for triggering/initiating adaptation (i.e. *initiation group*), whereas the other group contains metrics that are only required for making adaptation decisions (i.e. *non-initiation group*). Note that the categorisation is not necessarily exclusive meaning that metrics that are classified into the *initiation group* might also serve some purpose in the adaptation decision making.

Ideally, metrics in the *initiation* group should always be provided to the local adaptation engine at every measurement in order to let it decide whether adaptation is required. On the other hand, *non-initiation* metrics do not need to be delivered to the adaptation engine until adaptation has been triggered, i.e. the engine is ready to make adaptation decisions. In effect, since adaptation is generally triggered by changes in the execution environment (i.e. resource metrics) instead of the application behaviour (i.e. software metrics), software metrics are only delivered to the local adaptation engine when required (i.e. during decision making).

On the other hand, not only do newly collected resource metrics need to be propagated to the local adaptation engine for triggering purposes, they also need to be delivered to remote adaptation engines in order to facilitate decision making on remote machines. This requirement exists because the proposed adaptation algorithm (presented in section 3.2) requires resource metrics concerning destination (i.e. remote) hosts for predicting performance improvement. Specific criteria are used to reduce the overhead of these inter-host propagations, as described in section 4.2.3.

Since exchanging metrics between hosts (e.g. through multicasting) is non-trivial due to the possible overloading of the network as well as the ability of the individual node to receive the metrics, in this work, metrics are delivered to remote adaptation engines via a centralised context server [16]. In this approach, collected resource metrics are propagated from the participating host managers to the context server. At a later stage, an adapting host retrieves the metrics (of other nodes) from the context server using the pull mechanism, which as mentioned in section 2.4.3, offers the benefit of deferring the delivery of metrics until requested, thereby preventing unnecessary data transfer.

Note that the pull mechanism is also applied for the delivery of software metrics (from the containing runtime to the local adaptation engine) since these metrics are not required until the decision making phase. In contrast to resource metrics, software metrics are not required for decisions made on remote hosts, and thus not delivered to the centralised context server.

4.2.3 Management Criteria

In the proposed metrics solution, criteria are used to specify the interval of timer-initiated measurement (e.g. for collecting resource metrics) in order to control the frequency of such measurement. Furthermore, depending on the characteristics of the deployed application, frequency-based criteria can be used for controlling expensive measurement such as that for collecting SSP, so that such measurement occurs every N^{th} opportunity as opposed to every possible opportunity (i.e. every method invocation).

Criteria are also used for disabling the propagation of software metrics which as explained in section 4.2.2, will instead be explicitly pulled upon request (by the local adaptation engine). In addition, criteria are used to control the delivery of resource metrics to the centralised context server, in which case, the criteria serve to improve communication efficiency by delivering a family of metrics as a collective group rather than individually. For example, HPU (Host Processor Usage) and HNU (Host Network Usage) are always delivered together (i.e. considered as the same family) because both are host-related metrics and are collected at the same interval (e.g. every 5 seconds).

By using criteria to control operations (e.g. delivery to the adaptation engine) that are specific to the adopted adaptation solution, switching to a different solution can be achieved with minimal effort. For maximum flexibility, the decision of whether adaptation should be triggered is also handled by a criterion (i.e. *initiation criterion*), which first checks the *absolute* and *relative* availability of the system resources (e.g. bandwidth, CPU) against the initiation thresholds configured by the application deployer. *Absolute* availability refers to the level of the availability of a particular resource, whereas *relative* availability refers to the difference in the availability between the current and the previous measurement (i.e. the degree of

change in availability). In either case (i.e. absolute or relative), the threshold can be defined either as a concrete value (e.g. 100Kbps) or as a percentage (e.g. 10%). Due to the heterogeneity of target platforms being a strong motivator for application adaptation, the use of percentage-based thresholds is generally more appropriate.

If resource availability is low such that the thresholds were reached, the *initiation criterion* then determines whether the relevant node has the privilege (in the form of a shared token or lock) for performing adaptation. The adaptation token/lock serves to ensure that adaptation is performed in a consistent state (i.e. no other host is adapting at the same time), which could otherwise negatively impact on the accuracy of the result/decisions. The lock is exchanged upon demand, in which case, the requesting node will wait until the lock has been released by an adapting node, i.e. a remote node that is holding the lock. Once the lock has been acquired, an adaptation thread is launched in order to promote concurrency by ensuring that decision making is carried out independently from the metrics management activities and application execution.

4.3 Implementation

In contrast to the implementation of adaptation algorithms, which as mentioned in Chapter 3 is straight forward, the implementation concerning the metrics management solution proposed in this chapter, presents technical issues which are to be addressed in this section. In particular, the discussion is concerned with how the collection, representation, and delivery of metrics required by the proposed adaptation algorithm, is implemented in a Java-based mobile object framework: MobJeX [147]. Note that although it is not the focus of this thesis to address the management of metrics required by the original algorithm, such functionality was also implemented (in MobJeX), in order to facilitate the empirical evaluation presented in Chapter 5. The involved implementation effort is minimal particularly due to the flexibility and generality of the solution as well as due to the similarity between the original and the proposed algorithms.

To begin, a brief description of the MobJeX framework is provided as follows. MobJeX consists of two distinct support components for application adaptation: *offline* and *online* support. The *offline* support deals with compile-time injection of adaptation capabilities into an ordinary Java application, as discussed further in Chapter 7, which addresses the automation of such injection. On the other hand, the *online* support, which is implemented as middleware containers (e.g. host managers, runtimes), manages runtime operations such as object migration. Note that prior to this work, the provided support (i.e. both offline and online) was limited in that there was not support for adaptation as well as complementary functionality such as management of metrics and injection of adaptation capabilities.

MobJeX allows application objects to migrate from one machine to another as long as each participating machine runs an instance of the host manager, which in MobJeX is known as a *service*. A *service* is the main contact point of an individual machine, thereby enabling discovery/communication with other services and with the context server which is located in a centralised middleware component called the *system controller*. As is the case with the host manager described in section 2.5.1, a service handles the life cycle of all the *runtimes* running on the host. In MobJeX however, a runtime has the role of managing *application managers*, rather than directly managing mobile objects as shown in Figure 4-6, which provides an example of relationships between all the components involved in a deployed MobJeX system.

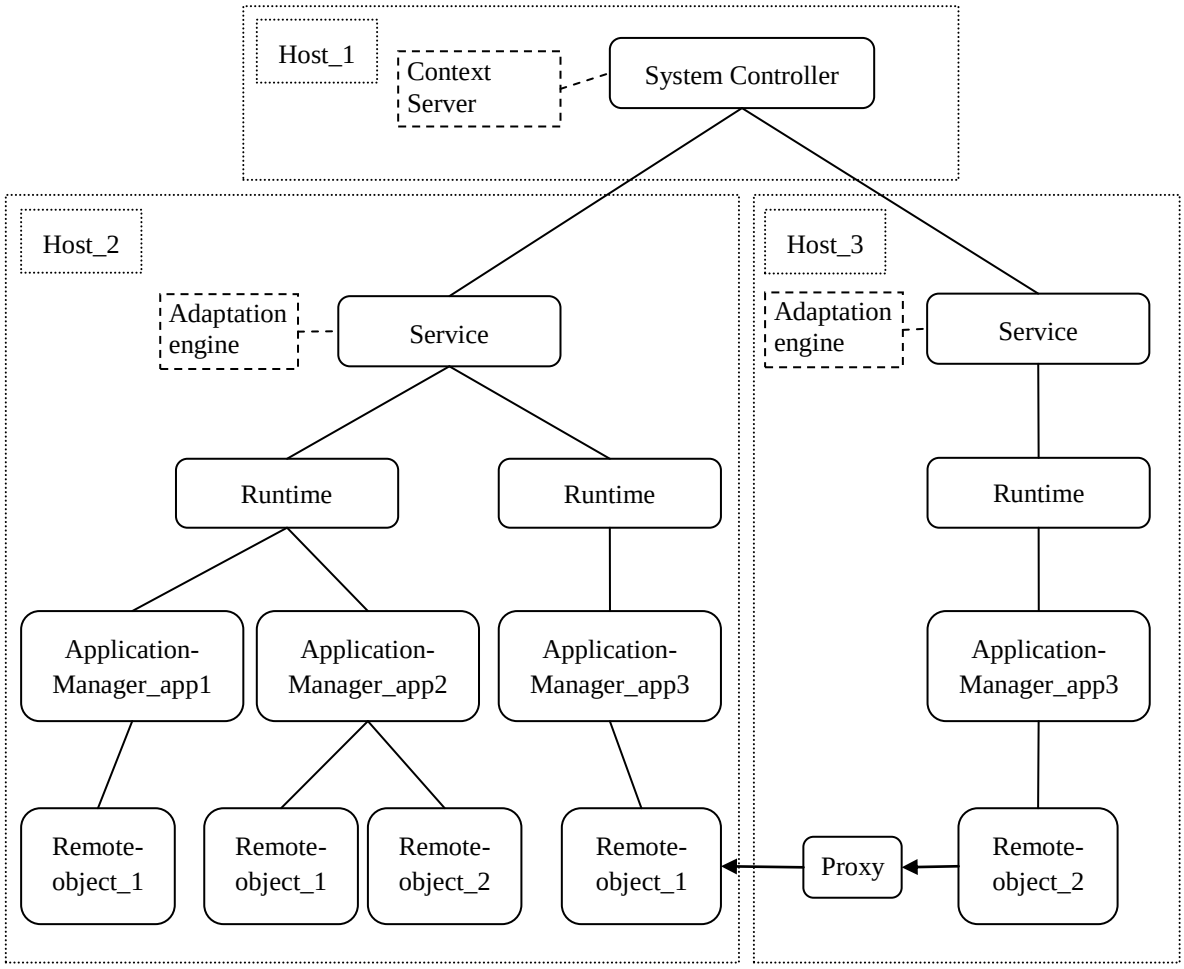

Figure 4-6. An Example of MobJeX Architecture

As the name suggests, the main role of an *application manager* is to administer a particular application, more specifically, the mobile objects (also known as *mobjects*) belonging to that application. The introduction of application managers allows multiple applications to execute independently (i.e. on separate threads, using different sets of classes) in the same runtime/process, thereby increasing efficiency at the expense of stability since a failed process will crash multiple applications. Note however that this flexibility is not exploited by the

proposed adaptation algorithm, since the algorithm as well as some of the adopted metrics such as RPU (Runtime Processor Usage), assume that the represented runtime is equivalent to the application itself, as discussed in section 3.2.2.

As is the case with the solution based upon system modelling discussed in section 4.2, the relationships between MobJeX components mentioned above, which include *services*, *runtimes*, *application managers*, and *mobjects* are modelled as interconnected model entities. Similarly, the relationships between the different components also dictate how metrics are propagated. For instance, metrics are pushed from a mobject (i.e. mobile object) to its immediate parent (i.e. an application manager) which further propagates the metrics as discussed in section 4.3.2.

In order to support various adaptation solutions, different implementations of metric-related objects (e.g. metric containers and metrics) need to be provided and grouped according to their relevance to a particular algorithm since certain objects are not applicable to other algorithms (e.g. `IFMetric` is used in the proposed algorithm but not the original algorithm). As such, in order to facilitate the grouping of related objects, the instantiation of these objects is managed using the *abstract factory pattern* [60], in which a separate factory is implemented for each supported adaptation solution. Each factory is responsible for creating objects (e.g. metrics, metric containers, criteria) that are relevant to the targeted adaptation solution. Since these factories implement the same interface, switching to a different adaptation solution can be done simply by swapping the chosen factory object (assuming that the implementation of the replacement solution exists).

Implementation issues related to metrics collection, metrics delivery, management criteria, and metrics representation are addressed in section 4.3.1, 4.3.2, 4.3.3, and 4.3.4, with particular emphasis on portability, code maintainability, metrics accuracy, and runtime efficiency/scalability.

4.3.1 Metrics Collection

For maximum portability, as much as possible the chosen implementation techniques should utilise the functionality provided by Java (i.e. JVM and API) rather than that specific to certain platforms (e.g. operating systems). However, certain measurements such as those for obtaining host metrics are reliant on the underlying support provided by the OS (Operating System). For example, in the case of MobJeX, when running on Windows XP or later, resource information is acquired from the operating system using a native C/C++ API called “performance monitoring”. For consistency, the unit of the acquired information (e.g. in seconds) might need to be converted into a specific unit used by other metrics (e.g. in milliseconds). Although the measurement itself uses a non-portable native library, the initiation of the

measurement can be triggered using a standard Java timer, which is portable across various platforms.

The collection of metrics involving native support relies on Java Native Interface (JNI) to bridge the framework (i.e. implemented in Java) and the operating system (i.e. native code). The framework code applies the *bridge design pattern* [60] to accommodate the varying metrics (e.g. HPU, HNU) as well as the different implementations for the supported operating systems (e.g. Windows, Linux). This approach allows new resource metrics to be introduced while also enforcing that relevant functionality be implemented for all of the supported operating systems.

The above solution applies to most host-related resource metrics with the exception of HPC (Host Processor Capacity), the collection of which requires human involvement due to the need for obtaining the relevant measures in instructions-per-second, which is more universal/cross-platform than widely available information such as cycles-per-second, as argued in section 3.2.2.1. Note that such a requirement does not affect the automation of adaptation since as mentioned in section 4.2.1.1, HPC is considered as a static metric and therefore only needs to be collected once, prior to application execution. The collection is achieved by means of calibration, whereby the duration of the execution of a simple program/method (e.g. consisting of an empty loop) is measured. Note that the more variety of instructions included in the program/method (instead of just an empty loop), the more accurate the approximation is, due to heterogeneity. Next, the approximated total executed instructions (in this case, the loop count), are divided with the duration in order to obtain the number of instructions that the processor is capable of executing per time unit (e.g. second).

Certain runtime-related resource metrics such as RMU (Runtime Memory Usage) and RMC (Runtime Memory Capacity) can be obtained from the JVM and thus imply a platform-independent implementation. On the other hand, a more complicated solution is required for collecting RNU (Runtime Network Usage) since even in the latest Java version (i.e. Java 6 as of this writing), the support for directly obtaining the network usage of a runtime is not available. The adopted solution involves replacing the default sockets used by Java RMI (Remote Method Invocation) with custom sockets for measuring the incoming/outgoing bytes that are transferred to/from the runtime. The feasibility of this solution relies on the fact that all the network communications that are performed by a MobJeX runtime, including those performed by the mobjects (mobile objects) executing inside the runtime, are achieved using Java RMI. While framework functionality (included in the middleware and the injected capabilities) can be enforced such that RMI is always used for remote communication, the application itself (which was developed independently) may use a different technique (e.g. communicating directly through sockets), in which case, the measured network usage will be inaccurate, i.e. lower than the actual usage.

The custom sockets (i.e. byte-measuring sockets) are created transparently from the RMI framework using the *factory method design pattern* [60]. The pattern is adopted in RMI to allow the specification of user-defined client and server socket factories, which are responsible for creating the custom client/server sockets. As shown in Figure 4-7, a client socket is used to monitor the outgoing traffic of the runtime, whereas a server socket monitors the incoming traffic. The results of the monitoring (i.e. in byte lengths) are accumulated and temporarily stored in the relevant metric object (i.e. `RNUMetric`), which at every fixed interval, will divide the accumulated number with the interval in order to obtain the latest sample of RNU.

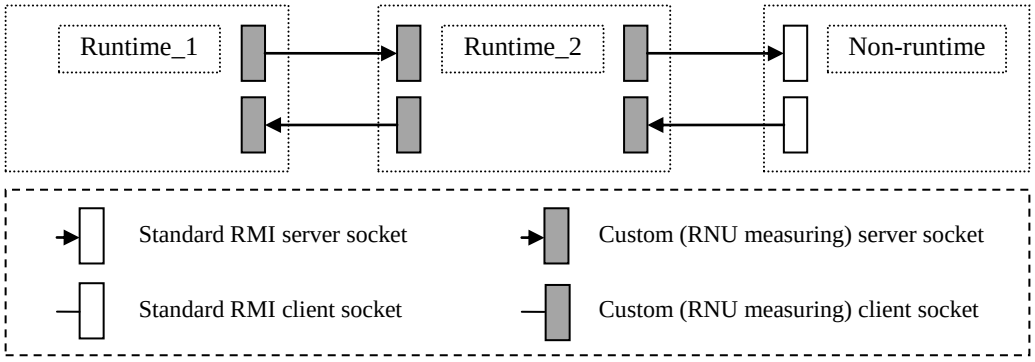

Figure 4-7. RNU Collection Using RMI Socket

As illustrated in Figure 4-7, a decision must be made regarding which type of socket (i.e. standard or custom socket) is used depending on the components involved in the communication. Custom sockets are used only for runtimes and their child components including application managers and mobjects (since the aim is to measure the network usage of runtimes). On the other hand, non-runtime components such as services (i.e. host managers) and system controllers use a standard RMI socket.

Since the collection of RNU utilises standard functionality provided by the JVM, the solution should be applicable to various JVMs and different operating systems. In comparison, the collection of RPU (Runtime Processor Usage) is platform-dependent, since it requires access (i.e. through JNI) to the process information provided by the OS. A possible improvement to this solution is to instead use the thread-level processor usage information provided by Java Management Extensions (JMX). Even though the finer-grained information (i.e. at thread level) could be beneficial (as explained in section 3.2.2.2), the solution involves higher complexity due to the need for identifying all the execution threads that are related to a particular application. In practice, this proves difficult due to the transient nature of certain threads such as timer threads and as such this solution has not been implemented in MobJeX.

With regard to software metrics, the metrics required by the proposed solution are those related to certain mobjects and their constituent methods. These include SSO (Size of Serialised Object), OMS (Object Memory Size), PUT (Processor Usage Time), SSP (Size of Serial-

ised Parameters), and IF (Invocation Frequency). Disregarding their initiation (i.e. time-based versus application-based) and location (i.e. mobject versus method), SSO and SSP are similar in that the relevant measurement is performed using object serialisation. The measurement is platform independent because it only makes use of the standard serialisation functionality provided by Java, in which the measurement subject (i.e. the measured object) is serialised through a custom stream object, which has the role of keeping track of the total number of bytes passing through the stream. Note that any other objects directly referenced by the measured object (i.e. not via proxies) are also included in the measurement.

In the case of SSO, the measurement subject is a mobject, whereas in the case of SSP, the subject includes zero or more objects depending on the communicated messages, i.e. the parameters and return values. In fact, the measurement subject of SSP never includes a mobject due to the constraint in remote communication in which a proxy should be passed (as parameters or return values) in place of the proxied object (i.e. mobject).

PUT and OMS are similar in terms of the portability of the measurement techniques since both require certain support from the JVM that is not universally available. The collection of PUT is achieved using Java Management Extensions (JMX) technology, whereas obtaining OMS requires certain instrumentation functionality provided by the JVM Tool Interface (JVMTI). Both technologies are only available in newer versions of JVM (i.e. Java 5 or above), but are not supported in the older versions or in cut-down implementations such as those targeted for mobile devices (e.g. IBM J9), in which case alternative techniques would be applied as follows.

The alternative for collecting OMS, which involves using SSO as a substitute metric, is less favourable than the main approach which uses JVMTI for two reasons. Firstly, SSO, which reflects the serialised size of the object, only provides an approximation of the in-memory size (OMS). Secondly, the tasks involved in measuring the serialised size of an object are less efficient than those involved in measuring its memory size. Note that the substitution also works the other way, meaning that in the presence of the required JVMTI support, OMS could be used as a substitute to SSO as a trade-off between efficiency and accuracy.

In the case of PUT, a native implementation similar to that used for collecting RPU can be used (in the absence of JMX support). This produces less accurate results compared to the JMX-based implementation because the measurement result reflects the usage of the entire runtime/process rather than the executing thread (i.e. the thread in which the method executes). Nevertheless, the inaccuracy will not have much impact on the adaptation of sequential applications due to the minimal resource contention from other threads, as mentioned in section 3.2.2.2.

Other than the issues that have been addressed in section 4.2.1.3, the implementation of the IF (Invocation Frequency) collection is fairly straight forward, except for a potential scal-

ability issue with the application of a timer-based approach. For instance, assigning a timer to exclusively handle the collection of IT concerning a particular method could result in a large number of timer threads depending on the size (i.e. number of methods) of the application. Such an overhead was in fact identified during the preliminary runs of the empirical overhead evaluation presented in section 5.4. Consequently, the preferred approach, which was subsequently implemented as part of this work, is to employ a pool of shared timers, each of which handle a certain number of methods (e.g. 100 methods). The use of multiple timers (which run in separate threads) serves to distribute the load in order to reduce the average waiting time of the involved operations. This solution can be further extended to support other timer-based metrics such as OMS and SSO, as long as the relevant measurement occurs at the same interval.

4.3.2 Metrics Delivery

Since resource metrics are exchanged between adaptation engines (i.e. services) through the centralised context server (i.e. system controller), a mechanism to allow the discovery of the system controller and the participating services, is required. At present, MobJeX only supports a manual discovery mechanism whereby the application deployer should either provide the location of the services to the context server or provide the individual services with the location of the context server.

The adopted *push* mechanism (e.g. for delivering software metrics to the local adaptation engine) is implemented using a queue-based approach, in which new metrics are appended to a queue for delivery. The queued metrics are then retrieved by a dispatcher, which propagates the metrics one at a time in a first-come-first-serve manner. The dispatching operation runs in an independent thread in order to minimise the performance impact of metrics delivery (e.g. caused by communication overhead) on application execution. In comparison to the *push* solution, the *pull* implementation is straight forward as it is achieved via direct remote method calls using RMI.

In the *fine-grained* delivery approach (described in section 4.1.2), delivered metrics should be accompanied by certain metadata in order to facilitate the reconstruction of missing information (e.g. the relationship between metrics, metric containers and model entities). At the minimum, the metadata should consist of the following information (or something equivalent): 1) a URI (Uniform Resource Identifier) used for identifying the measured component (e.g. a specific object), 2) a class reference (i.e. `java.lang.Class`) to identify the measured attribute (e.g. SSP), and 3) (if applicable) a method reference (i.e. `java.lang.Method`) to identify the method in which the metric was collected, for the pur-

pose of calculating metrics concerning the same method (e.g. multiplying IF and SSP of a particular method to get the interaction intensity of that method).

4.3.3 Management Criteria

All the criteria mentioned in section 4.2.3, which include those for controlling metrics collection and metrics delivery, have been implemented in MobJeX as normal Java classes which implement certain predefined interfaces. The implemented criteria also include other basic criteria such as the *always* criterion which indicates that the operation should always be carried out or the *never* criterion which permanently prevents the execution of the relevant operation.

The criteria approach was designed and implemented in such a way that different criteria can be selectively used for different components (e.g. individual mobjects) and different operations. For instance, the collection of PUT might use an *always* criterion due to its relatively low overhead, whereas SSP might apply a *frequency-based* criterion to limit the measurement frequency to occur once every 5 invocations. A factory is also provided to control the instantiation of criteria (in terms of determining which criterion is used for which component/operation) based on the configuration specified by the application deployer.

4.3.4 Metrics Representation

The implementation of the proposed metrics solution allows different strategies (e.g. moving average, standard average) to be used for representing various metrics. This is achieved by creating a representation object for every metric object (e.g. `RPUMetric`), which as mentioned in section 4.1.1, serves to encapsulate the functionality related to the corresponding metric (e.g. RPU). Nevertheless, in the specific case of MobJeX, an exception is applied to the implementation of SSP and IF, which for efficiency reasons, is combined into one single class due to the cohesiveness of the related operations (e.g. collection, calculation, delivery).

The different types of representation described in section 4.1.4 have been implemented in MobJeX. These include: basic representations with no temporal information (i.e. only uses the last measured value); representations using standard average (i.e. arithmetic *mean*); and representations using Exponentially Weighted Moving Average (EWMA). The implementation of these representations is efficient both computationally and space-wise, since none of them requires the maintenance of metrics history.

Since the different implementations follow the same interface (i.e. method signatures), they can be swapped with each other without affecting the manner in which the adaptation engine processes/uses the metrics. In addition, a *factory method pattern* [60] was used to cen-

tralise the creation of the representations, thus enabling easy switching between different sets of representations.

The evaluation of the metrics management solution presented in this chapter as well as the adaptation algorithm proposed in Chapter 3, will be presented in the next chapter, which focuses on aspects such adaptation effectiveness, efficiency, and scalability.

Chapter 5. Adaptation Evaluation

As discussed in section 4.3, both the *original* and the *proposed* adaptation algorithms (presented in Chapter 3) as well as complementary functionality, such as that related to metrics management and adaptation initiation (as addressed in Chapter 4), have been implemented in a mobile object framework MobJeX in order to facilitate the empirical evaluation presented in this chapter. Sections 5.2-5.4 present the experiments undertaken using the framework to 1) validate the behaviour of the proposed decision making algorithm, 2) evaluate its effectiveness in terms of improving the performance (i.e. execution duration) of the adapted application, and 3) evaluate the overheads of decision making and metrics management in terms of their effect on application performance and system resource utilisation (i.e. processor, network, memory).

Note that the rest of this chapter is structured such that experimental materials and procedure that are common to certain experiments are presented in a section preceding the discussion of the experiments. On the other hand, additional materials and procedure which are specific to a particular experiment are presented as a sub-section of the relevant discussion. The experimental materials and procedure applicable to all experiments are presented in section 5.1 below.

5.1 Common Experimental Materials and Procedure

The experimental materials used to execute all experiments described in this chapter consist of machines with quad-core CPUs (i.e. Intel Q9550) clocked at 2.83 GHz, connected via 100 Megabits per second (Mbps) Ethernets. However, for the majority of the experimentation, multi-core capability was disabled through BIOS settings, because the existence of multiple cores reduced the predictability and reproducibility of experiments as explained further in section 5.3. The exception to this was the last phase of the experiment presented in section 5.3.4 and the overhead evaluation presented in section 5.4, in which multi-core capability was enabled in order to show its performance implication in terms of enabling parallel execution of application-specific as well as adaptation-related operations (e.g. decision making, metrics delivery) on multiple cores.

Each machine ran 32-bit Windows XP Service Pack 3 and all test applications were developed in Java and executed using the JVM from the Sun Java SDK version 1.6.0_17 distribution (the newest stable version as of this writing). Although disabling the Just-in-time (JIT) compilation feature of the JVM theoretically allows more consistent execution, this was not done due to an identified bug, which as documented in [168], would result in fluctuating execution duration over successive execution runs. Consequently, the experiments were con-

ducted using the default JIT compilation mode, in which methods are compiled into native code at runtime in a selective manner, i.e. depending on how many times the relevant methods execute.

Rather than containing real-world functionality, test applications were written such that they exhibited deterministic characteristics (e.g. pre-defined interaction behaviour) in order to facilitate the relevant validation or evaluation (e.g. objects exhibiting intensive remote communication should favour machines with more network bandwidth). The specific characteristics are discussed in detail in the relevant section (i.e. 5.2.2 or 5.2.3).

The following experimental procedures are common to all experiments. Firstly, the test application (used in a particular experiment) was converted into an adaptive application through capability injection, which as addressed in Chapter 7, involved code transformation. Such injection was performed at deployment time using the support from the MobJeX framework developed as part of this thesis (as discussed in Appendix C). On the other hand, the execution of the injected application was facilitated by middleware components (also a part of the framework), which as mentioned in section 2.5.1, includes services (i.e. host managers) and runtimes.

Prior to execution, adaptation behaviour was configured according to the specific characteristics of the experiment. The performance improvement indicator, which as introduced in section 3.1, refers to the extent to which adaptation decision making should favour performance improvement as the goal of adaptation, was set to a maximum value of 1. On the other hand, other indicators, which include those related to balancing of processor, network, and memory load, were set to 0 to indicate that the sole objective of the adaptation concerned in this experimentation is performance improvement.

The migration cost calculation formula (which was addressed in section 3.2.2.3) was configured such that migrating a 1MB object over a 100Mbps network link would offset performance improvement by 2%. Such an offset was chosen since it was sufficiently large that the difference in migration priority (i.e. sequence) between objects of small and large sizes was noticeable, as demonstrated in section 5.2.2.

The Host Processor Capacity (HPC) metric, which is required for adaptation, was collected using the calibration technique described in section 4.3.1, which revealed that the CPUs (or more precisely each CPU core) used in this experimentation, can execute up to approximately 700 millions of Java instructions per second.

5.2 Decision Making Behaviour Validation

The experiments in this section aim to verify the correctness of the object placement behaviour of the proposed algorithm in terms of conformance to a set of *axioms*, which express the

behaviour of the proposed algorithm in cases where it is expected to differ from the original due to the improvements (e.g. changing existing metrics, adding new metrics) applied to the original score calculation formulas, as presented in section 3.2.

The experiment concerning decision making pertaining to the processor and memory availability of machines is presented in section 5.2.2, whereas validation involving network availability is conducted as a separate experiment (i.e. section 5.2.3) in order to isolate the associated complexity, thus allowing more accurate analysis of decision making behaviour.

Although not the primary focus, the first experiment (section 5.2.2) also facilitates the verification of the correctness of adaptation-related functionality, e.g. to ensure that adaptation is: 1) triggered at the appropriate time (e.g. when resource availability is low) and 2) performed exclusively (by a single adaptation engine at any one time) in order to prevent conflicting decisions, as described in section 4.2.3. However, such verification will not be discussed further since not only is it implementation-specific, but the associated process (which involves analysis of log statements) is straight forward.

The experimental procedure which is common to the experiments presented in sections 5.2.2 and 5.2.3, are discussed in the following section.

5.2.1 Experimental Procedure for Behaviour Validation

In order to explicitly demonstrate the behavioural difference between the original and the proposed algorithms, each of the concerned experiments (presented in sections 5.2.2 and 5.2.3) was executed in two adaptation modes: one using the original algorithm and another involving the proposed algorithm.

Resource metrics were pre-scripted in order to control the characteristics (in terms of resource availability or usage) of the execution environment, whereas software metrics (i.e. those related to application characteristics) could be automatically collected without scripting (using the metrics management solution presented in Chapter 4) since application behaviour was controlled by the test application having been purposely written to exhibit specific characteristics (e.g. execute for X seconds) as elaborated in sections 5.2.2.1 and 5.2.3.1.

Timer-based measurement, which applies to the collection of resource metrics (the values of which were pre-scripted in this experiment), was set to be performed every 1 second, thereby implying adaptation (which is triggered by the detection of low resource availability) would be executed at most once every second. This frequency is sufficiently high for adaptation to be executed at least once before the application finishes executing.

Certain variables (as listed below) were controlled in order to facilitate the prediction and validation of specific decision making behaviour (for example the sequence in which objects would get migrated) in a reliable and reproducible manner.

- Instead of allowing adaptation to be performed at any point in application execution (which is the case in regular execution since decision making runs independently of the application thread), synchronisation was used to ensure that prior to each occurrence of decision making, all mobile objects had been executed a pre-defined numbers of times (which might vary between objects). This was done so that decision making behaviour could be accurately assessed because object characteristics were known at the time adaptation decisions were made.
- The collection of software metrics involved the most comprehensive configuration, in which metrics were collected at every possible opportunity rather than selectively (as explained in section 4.1.3), since the focus was on traceability (e.g. X invocations imply exactly X occurrences of software-metrics collection) as opposed to efficiency.
- Metrics temporality (section 4.1.4), which was concerned with weighting the importance of metrics depending on their age, was ignored/disabled since this would otherwise present additional variables which would unnecessarily complicate behaviour prediction.

The adaptation initiation threshold was specified such that adaptation was triggered when the availability of a particular resource was equal to or less than 80%. Such a threshold was chosen because it is relatively low compared to the ideal condition of 100% availability that adaptation might bring performance benefit. At the same time the threshold is not so low that adaptation agility (which refers to how fast the application adapts to changes) is compromised. Such configuration was shown to be appropriate in the experiment presented in section 5.3, which suggested that there was no notable performance improvement (gained from adaptation) when CPU availability was higher than 80%.

The adaptation improvement score threshold for the migration of a particular object (as explained in section 3.2.2.4) was set to 5%, thereby implying expected minimum improvement of 25-35% assuming that all mobile objects of the test applications (which comprise 5-7 mobile objects) are adapted/migrated. Such a threshold was chosen because 25-35% represented sufficiently significant (and thus realistic) expected improvement, although in practice, the gained improvement is less (i.e. < 25-35%) due to the associated overhead (of decision making, metrics management, etc.) and due to the fact that, some objects might not migrate (thus not gaining 5% improvement per object) because of their low resource requirements. The chosen threshold was also low enough that significant objects (i.e. those with relatively high resource requirements) would eventually be offloaded to other machines (with better conditions), thereby facilitating comprehensive analysis of decision making behaviour in terms of migration priority (i.e. the sequence in which objects migrate) and destinations (i.e. to which machine a particular object migrates).

The correctness of the tested behaviour was verified through analysis of the produced log statements (e.g. calculated decision-making scores) as well as observation of migration

events via a graphical administration interface which shows up-to-date information (e.g. location, state) about objects and machines.

5.2.2 Validation Pertaining to Processor and Memory Availability

This experiment aims to demonstrate correct object placement in a scenario involving objects and machines with heterogeneous processor and memory utilisation (network is covered in section 5.2.3). Not only does this experiment demonstrate that the proposed algorithm conforms to the axioms defining the expected object placement behaviour (listed below), but it also provides comparison with the original algorithm, as analysed in section 5.2.2.2.

`Axiom 1`: When comparing two runtimes as a possible destination for an object, assuming equal resource availability between machines, the runtime with higher resource utilisation is favoured because in sequential applications, the resource utilisation of a runtime (which reflects the utilisation of the application objects currently managed by the runtime) contributes to the overall amount of resources that are available to the application, as explained in section 3.2.2.2. Note that this axiom assumes the case in which each runtime manages a single rather than multiple applications, because the support for the latter is not commonly available in existing frameworks, as mentioned in section 3.2.2.2.

`Axiom 2`: Objects never migrate to a runtime which does not have sufficient memory despite its superiority over other machines with regard to the availability of other resources (e.g. CPU). This is due to the consequential execution failure discussed in section 3.2.

`Axiom 3`: Decision making favours migration of objects with CPU-intensive methods, regardless of execution duration which is affected by non-CPU operations, such as thread suspension, I/O accesses, etc., as addressed in section 3.2.2.1.

`Axiom 4`: The higher the cost of migrating a given mobile object (i.e. larger size), the less likely the object will migrate, as explained in section 3.2.2.3.

5.2.2.1 Experimental Materials

The specific materials used in this experiment, which include a synthetic test application and machines of particular configuration, are described below.

- Table 5-1 describes the unique characteristics of each of the five mobile objects in the test application.

Mobile Object	CPU Usage Intensity	In-memory Size
M1	High	Small
M2	High	Large
M3	Low	Small
M4	Low	Large
M5	Vary gradually (low to high)	Small
Where:	High = 100% CPU usage Low = 0% CPU usage	Large = 10 MB Small = 1 KB

Table 5-1. Objects with Various CPU and Memory Requirement

Mobile objects M1-M5 are executed in sequence, for a number of iterations (i.e. 10) in order to allow sufficient time for the application to adapt. In each iteration, each object executes for the same amount of time (i.e. 1 second), but the intensity of the execution varies in terms of CPU usage (as shown in Table 5-1). This is done in order to facilitate the validation of the decision making behaviour stated in axiom 3. CPU usage intensity is controlled by interleaving two operations: the execution of a sequence of CPU-intensive operations, and the suspension of the executing thread (i.e. switching the thread into sleep/idle mode). As such, the longer the execution of the operations (relative to the thread sleep time), the higher the resulting CPU usage intensity (of a method/object).

Note that execution frequency was kept the same for all objects to avoid affecting network resource usage, which is instead the focus of section 5.2.3. Furthermore, all of the methods are void (no parameters or return values), again to avoid affecting network resource usage. In terms of size, application objects are split into two groups. One group represents large objects with the size of 10 MB, whereas the other represents smaller objects (i.e. 1 KB). Rather than emulate real-world execution scenarios, in which 1 KB is not necessarily considered small, such variation serves to facilitate the validation of behaviour concerning the prevention of migration to memory-constrained target runtimes (axiom 2) and prioritising migration based on estimated cost derived from object size (axiom 4).

- Table 5-2 describes the unique characteristics of the machines involved in this experiment.

Machine	Host Processor Availability (HPA)	Runtime Processor Utilisation (RPU)	Runtime Memory Availability (RMA)
S	Vary (high to low)	High	High
T1	Low	High	High
T2	High	Low	High
T3	High	High	Low

T4	High	Medium	High
Where:	High = 50% of HPC Low = 25% of HPC	High = 50% of HPC Medium = 35% of HPC Low = 25% of HPC	High >= 10MB Low < 10MB
	HPC = Host Processor Capacity		

Table 5-2. Execution Environments with Varying CPU and Memory Availability

The source machine S (the machine in which all objects of the test application initially executes) is connected to target machines: T1-T4. The resource availability of these machines is varied in specific ways (as shown in Table 5-2) in order to present a meaningful scenario for validating the concerned adaptation behaviour as described below.

As shown in the second column of Table 5-2, the CPU availability of the source machine is configured to gradually decrease during application execution, in order to help verify that adaptation is triggered accordingly. The gradual decrease also serves to provide sufficient time for the application to execute and metrics to be collected before adaptation gets triggered. In contrast, each target machine is configured with a fixed (i.e. unchanged) CPU availability in order to enable more accurate prediction of and more reproducible decision making behaviour. CPU availability is varied through the manipulation of CPU usage, which is achieved using a technique similar to the aforementioned approach for varying the CPU usage intensity of objects, i.e. by interleaving CPU-intensive operations and thread suspension.

As can be seen from the third column (of Table 5-2), the manipulation of CPU usage also applies to the runtime executing on each machine, in order to enable validation of the behaviour stated in axiom 1. Runtime CPU usage for T2-T4 is varied in order to demonstrate that the proposed algorithm favours one over the other, despite the equal CPU availability of the relevant machines (i.e. High HPA). Note that the variation of RPU differs from real-world scenarios, in which case the CPU utilisation of a runtime is directly influenced by objects (of the test application) executing on the runtime. Additionally, in practice, since all objects initially execute on the source machine/runtime, the CPU usage of target runtimes should be insignificant (or even zero if middleware management tasks are not accounted for), which is not the case in this experiment since as mentioned in section 5.2.1, resource metrics are pre-scripted.

The last column of Table 5-2 shows the different memory availability of runtimes, which serves to verify that in the proposed algorithm, objects should never migrate to runtimes that do not have sufficient available memory, as stated in axiom 2.

5.2.2.2 Analysis of Decision Making Behaviour

The outcome of the experiment conducted using the proposed algorithm demonstrates that the exhibited behaviour conforms to all the previously stated axioms. Table 5-3 summarises the outcome by outlining the relative sequence in which objects are migrated (column 1) and the destination of each object (column 3). The following discussion, the primary purpose of which is to explain the experimentation outcome, is complemented with a brief description of the behaviour that is exhibited when the original algorithm is used.

Order	Object	Machine
1	M1	T3
2	M2	T4
3	M5	T3
Never migrate	M3	N/A
Never migrate	M4	N/A

Table 5-3. Object Placement Based on CPU and Memory Availability (Proposed Algorithm)

As stated in *axiom 1*, the destination T3, which has high HPA and RPU, is perceived to have the highest CPU availability (i.e. essentially 100% assuming there is no interference from background processes). This is verified through the migration of CPU-intensive objects with low memory requirement (i.e. M1 and M5) to T3. In comparison, using the original algorithm, objects are migrated to either T2, T3, or T4, because these destinations are perceived to have the same CPU availability due to the limitation described in section 3.2.2.2, in which internal load (i.e. the load contributed by the application itself or its runtime) is ignored. Such a limitation results in suboptimal decision making and overhead caused by unnecessary (or even ping-pong) migrations, as demonstrated later in the results from experiments involving live adaptation (i.e. using real metrics) presented in section 5.3.2 and 5.3.3.

The behaviour stated in *axiom 2* is validated through the fact that M2, which has high memory requirement, is migrated to a runtime having sufficient free memory (i.e. T4) despite its slightly inferior CPU availability (when compared to T3). In contrast, memory constraint is not considered in the original algorithm and thus all objects (including M2) are migrated to either T2, T3, and T4 for the same reason mentioned in the preceding paragraph. Since runtime memory availability is only simulated, execution failure (due to insufficient memory) is not exhibited in this particular experiment.

According to *axiom 3*, objects of lower CPU intensity should have lower migration priority. This is demonstrated by the fact that M3 and M4 are not migrated, but rather remain in the source machine. Furthermore, the migration of M5 occurs at a later stage since its CPU requirement, which gradually increases, is not yet significant early in the execution. In contrast, in the original algorithm, all objects (including M3 and M4) are migrated because they

are perceived to be of the same significance due to their equal execution duration. Furthermore, the migration occurs in an implementation-specific order (i.e. the order in which objects get registered to the adaptation engine) as opposed to a more meaningful sequence (i.e. according to their significance with regard to how much improvement can be gained).

The conformance to *axiom 4* is demonstrated by the fact that M2, which is of larger size (than M1), migrates after M1. Such behaviour, which is due to the implied higher migration cost (thus having lower migration priority), is not exhibited by the original algorithm, which as mentioned, migrates objects in an implementation-specific order.

5.2.3 Validation Pertaining to Network Availability

This experiment aims to test object placement behaviour in a scenario involving varying network bandwidth availability and object interaction intensity. The following axioms state the network-related behaviour that should be exhibited by the proposed algorithm.

`Axiom 1`: The migration of an object that is CPU-bound favours machines with higher processor availability unless the object has high network bandwidth demand, in which case machines that can offer higher communication bandwidth are favoured. Note that communication bandwidth availability is defined as the lower availability of the two communicating machines since this represents the bottleneck of communication.

`Axiom 2`: Objects exhibiting a higher degree of interaction with each other, i.e. higher Invocation Frequency (IF) and larger Size of Serialised Parameters (SSP), are more likely migrated to the same machine (than those with a lower degree of interaction).

`Axiom 3`: Varying the outgoing IF of a method, changes the ratio between its interaction and execution intensity, whereas varying its incoming IF does not. This is because outgoing IF only affects interaction intensity, whereas incoming IF affects both execution and interaction intensity as explained in section 3.2.1.

`Axiom 4`: In the context of remote outgoing communication, varying IF by a particular factor is nearly although *not* equivalent to changing SSP by the same factor. More specifically, increasing IF results in higher interaction intensity because of the additional information (e.g. metrics) that is communicated in each invocation, as explained in detail in section 3.2.2.1.

5.2.3.1 Experimental Materials

The specific materials used in this experiment, which include a synthetic test application and machines of particular configuration, are described below.

- Figure 5-1 illustrates the specific interaction behaviour exhibited by individual mobile objects of the test application, wherein mobile objects are accessed (through method invocation) by a stationary object S1 for a number of iterations (i.e. 10) in order to allow sufficient time for the application to adapt.

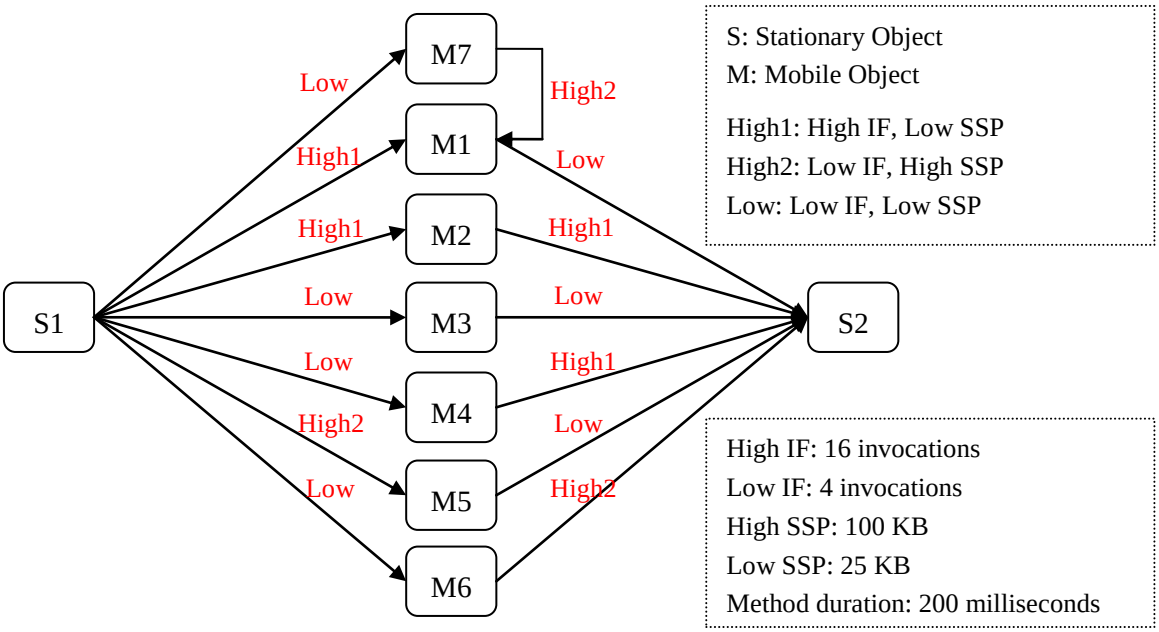

Figure 5-1. Objects with Various Interaction Characteristics

As mentioned in section 2.5, stationary objects are applied in real-world applications for objects such as those representing database connectors or GUI components. However, the role of stationary objects in this experiment, is to prevent a straight-forward adaptation outcome in which all mobile objects are migrated to a single machine (for the purpose of minimising network communication), thereby allowing the behaviour stated in axiom 1 to be validated.

Despite executing the same CPU-intensive operation (for 200 milliseconds), the overall duration in which each mobile object executes may vary in each iteration, depending on how many times the object is accessed by the *source* object S1. The duration may vary further since each object accesses a *target* object (which might be located remotely) for a particular number of times during its execution. In most cases, the target object is another stationary object S2, except for the case of M7, which as depicted in Figure 5-1, accesses M1 instead, in order to demonstrate conformance to axiom 2.

The interaction intensity between a particular mobile object and its *source* as well as *target* objects is varied in terms of number of invocations (which correlates with IF) and SSP in order to assist the validation of the behaviour stated in axiom 3 and axiom 4. The exact number of invocations and SSP of individual objects (which are shown in Figure

5-1) were chosen using a trial-and-error approach, the purpose of which is to find a good balance between interaction and execution intensity.

For instance, the value 200 ms was chosen for Processor Usage Time (PUT) because this is considered relatively low compared to the average interaction intensity of mobile objects in this experiment, thereby allowing the impact of object interaction on decision making to be observed. Nevertheless, it is important that PUT is still high enough that adaptation (and the resulting migration) will bring performance benefit. Another consideration is that IF and SSP were specified such that: `High IF × Low SSP == Low IF × High SSP` (as detailed in Figure 5-1), in order to show that although the two interaction characteristics are mathematically equal, decision making concerning these characteristics exhibit a different outcome as stated in axiom 4.

- Table 5-4 describes the characteristics of the machines involved in this experiment.

Machine	Host Processor Availability (HPA)	Host Network Availability (HNA)
S	Vary (100% to 0%)	Medium
T1	High	Low
T2	Medium	Medium
T3	Low	High
Where:	High = 80% of HPC Medium = 65% of HPC Low = 60% of HPC	High = 75% of HNC Medium = 50% of HNC Low = 25% of HNC

Table 5-4. Execution Environments with Varying CPU and Network Availability

As is the case with the first experiment, the processor availability of the source machine S, which is the machine where all mobile objects are initially located, is set to gradually decrease in order to let metrics be collected before adaptation gets triggered. On the other hand, the target machines T1-T3 are configured to have unchanged CPU availability in order to promote predictability and reproducibility.

CPU and network bandwidth availability across target machines are varied in certain ways (as shown in Table 5-4) in order to demonstrate conformance to the established axioms. For example, “low” CPU is defined to be slightly lower than “medium” in order to show that T3 is effectively a slower execution platform than T2 despite its superiority in bandwidth availability, which does not offer additional benefit due to the bandwidth limitation of other machines, as stated in axiom 1. As this fact is not accounted for in the original algorithm, a different behavioural outcome between algorithms is expected. Otherwise, had “low” CPU been set to be significantly lower than “medium”, this behavioural difference might not be noticeable.

5.2.3.2 Analysis of Decision Making Behaviour

The summary of the experimentation outcome (using the proposed algorithm) is presented in separate tables based on the destinations (i.e. either T1 or T2) of mobile objects, because the sequence of object migration is deterministic only when the same destination is concerned, due to the difference between the two destinations in terms of the availability of resources (i.e. CPU and network) which are not directly comparable (e.g. X amount of CPU and Y amount of network are not necessarily equal to X network and Y CPU). Due to the significant difference in how interaction intensity is calculated in the original and proposed algorithms (as explained in section 3.2.1), the established axioms, which primarily target the proposed algorithm, cannot be used as a baseline for comparison between algorithms and therefore, a general comparison between the two algorithms is instead provided at the end of this section.

Sequence	Object	Destination
1	M1	T1
2	M7	T1
3	M3	T1

Table 5-5. Migration of Objects to Target Machine T1

The conformance to *axiom 1* is demonstrated by the fact that objects that communicate intensively with stationary objects S1 and S2, which include M4-M6, are migrated to T2 (as shown in Table 5-6), because T2 is (correctly) perceived to be the most suitable destination due to its higher bandwidth availability compared to T1 and due to the fact that the extra bandwidth provided by T3 cannot be exploited (because the limited bandwidth of T1 and T2 is a bottleneck in the communication with T3).

Sequence	Object	Destination
1	M2	T2
2	M4	T2
3	M5	T2
4	M6	T2

Table 5-6. Migration of Objects to Target Machine T2

Conforming to the behaviour described in *axiom 2*, M7, which has a unique characteristic of being coupled with M1 (as opposed to S2), migrates to the same machine as M1 (i.e. T1). Analysis of log statements shows that the calculated improvement score of the relevant object-node pair (i.e. M7 to T1), increases considerably after the migration of M1 to T1, thereby causing the migration (of M7 to T1) to follow immediately.

The behaviour described in *axiom 3* is validated through the fact that M1 and M3, which differ only in the frequency at which they are accessed by the source object, migrate to the same machine (i.e. T1). In this case, both objects are perceived to be more execution (rather than interaction) intensive and furthermore varying their incoming IF (i.e. “High1” versus

"Low") does not affect the ratio between execution and interaction intensity. On the other hand, M2, which has the same incoming IF as M1 but higher outgoing IF (thus higher interaction intensity), migrates to a machine having higher network availability (i.e. T2). Note that although both M1 and M3 favour the same destination machine, their migration priorities are different. In particular, due to its higher execution frequency which results in higher execution intensity, M1 has a higher priority and thus migrates earlier.

The proposed algorithm conforms to axiom 4 due to the fact that M4, which differs from M6 only in that its outgoing interaction intensity is largely contributed by high IF (as opposed to SSP), migrates before M6. This is because of the expected behaviour stated in axiom 4, in which, compared to SSP, IF has higher impact on interaction intensity.

In comparison to the proposed algorithm, the original adaptation algorithm is not able to distinguish local from remote calls and thus assumes all of the recorded calls of an object to be remote calls. Such behaviour favours a machine having higher free network bandwidth. Consequently, using the original adaptation algorithm, all mobile objects are migrated to T3, which is perceived to be the most powerful machine due to its high network availability (which is incorrect as stated in axiom 1). Although such behaviour is less optimal than that exhibited by the proposed algorithm, its impact on application performance is minimal, due to another more severe limitation (in the original algorithm), which causes unnecessary migrations as demonstrated in the evaluation presented in sections 5.3.2 and 5.3.3.

5.3 Adaptation Effectiveness Evaluation

This section discusses the three experiments conducted to evaluate the overall effectiveness of the proposed adaptation solution, which is defined in terms of the impact (i.e. improvement) of adaptation on the performance (i.e. execution duration) of the adapted application. Consequently, not only does such an evaluation serve to assess of the quality of decision making (which affects the degree of performance improvement), but it also serves to demonstrate that the overheads of the involved operations (e.g. decision making and metrics management) are low relative to the gained performance improvement. Nevertheless, these adaptation overheads will be discretely evaluated in section 5.4 in order to provide better understanding of their characteristics (e.g. trends) as well as their impact on scalability.

The first and the second experiments, which will be presented in sections 5.3.2 and 5.3.3, apply configurations similar to the experiments concerning decision making behaviour validation presented in sections 5.2.2 and 5.2.3. This is because such configurations adequately cover various experimentation scenarios through the inclusion of objects and machines of different characteristics. Additionally, these experiments serve to demonstrate how the behav-

iour exhibited and analysed in the base experiments, affects application performance in live adaptation (using real metrics).

The third experiment (presented in section 5.3.4) extends the scope of experimentation by evaluating the effectiveness of the proposed adaptation solution in dynamically changing execution conditions, which include those related to the application itself as well as its execution environments. Such an evaluation complements the previous two experiments, which primarily address the heterogeneity of execution conditions as opposed to their dynamic nature, which is also a motivator for application adaptation.

All three experiments involve three distinct execution modes. In the first mode, a test application is executed in its original form, thus not involving additional operations such as object mobility, metrics management, and adaptation execution. In contrast, the other two modes involve adaptation performed using either the original algorithm or the proposed algorithm. These execution modes serve to show the difference in the effectiveness of the two adaptation algorithms as well as the extent of the resulting performance improvement/degradation (compared to the original non-adaptive application).

Note that the application used in the work [144] on the original adaptation algorithm (i.e. a Taxi Dispatching System) was considered but not used in this experiment, since it was deemed to be inappropriate for performance-based adaptation due to its low CPU utilisation. Such an application would not benefit from adaptation in a real scenario (despite the encouraging results shown in the original work) because the application of live metrics collection (which was ignored in the work) would incur runtime overheads, which consequently offset the little benefit gained by applications with low CPU demand.

5.3.1 Experimental Procedure for Effectiveness Evaluation

The execution of the first, second, and third experiments, which are discussed in sections 5.3.2, 5.3.3, and 5.3.4, applied the following procedure.

Two versions of the test application were used. The first was the adaptive version which was used for execution with either the original adaptation algorithm or the proposed algorithm, in order to compare their effectiveness. The second was the original non-adaptive test application, which provides a baseline for calculating application performance (which is a measure of adaptation effectiveness). More specifically, the effectiveness of a particular solution was calculated by subtracting *the duration of adaptive execution using the chosen solution*, from *the duration of non-adaptive execution*, as summarised in equation 1 of Figure 5-3. Adaptation effectiveness is also presented (in sections 5.3.2, 5.3.3, and 5.3.4) as a percentage, which as shown in equation 2, was calculated by dividing the result from the calculation de-

scribed in the preceding sentence, with the baseline duration (i.e. the duration of non-adaptive execution), then multiplied by 100%.

(1) $AbsoluteImprovement = NonAdaptive - Adaptive$

(2) $RelativeImprovement = \frac{EffectivenessTime}{NonAdaptive} \times 100\%$

Where:

AbsoluteImprovement = Adaptation effectiveness in milliseconds

RelativeImprovement = Adaptation effectiveness as a percentage

NonAdaptive = The duration of non-adaptive execution (in milliseconds)

Adaptive = The duration of adaptive execution using the original or the proposed algorithm (in milliseconds)

Figure 5-2. Adaptation Effectiveness Calculation

The collection and management of metrics required for adaptation (which was addressed in Chapter 4) was configured as follows. The measurement of time-based metrics (e.g. resource metrics) was performed every second, in order to cope with dynamic execution conditions. Such configuration, despite being less important in the first and second experiments (since dynamic execution conditions were not a concern), serves to represent a worst-case scenario since 1 second is the most frequent sampling rate supported by the underlying operating system (i.e. Windows XP), thereby incurring the largest possible overhead. Nevertheless, in order to represent a realistic scenario, appropriate criteria were used to moderate the overhead incurred by metrics management. In particular, expensive metrics were collected at every fifth opportunity, i.e. SSP at every 5^{th} invocation and SSO every 5 seconds. Furthermore, similar metrics were grouped and delivered (to the context server) collectively, as described in section 4.2.3, in order to reduce the associated overheads.

Since resource metrics were collected every one second, adaptation decision making is triggered at most once every second. However, in order to prevent the original algorithm from constantly migrating objects between machines (a behaviour which was observed in the preliminary execution of this experiment due to the limitation described in section 3.2.2.2), the original algorithm was complemented with a time-based workaround for preventing decision making unless 5 seconds had elapsed since the last adaptation decision. This workaround has a negative effect of lessening adaptation agility (i.e. the ability to quickly respond to changing conditions), but is a reasonable compromise considering it prevents constant migration of objects (i.e. every second). Note that this workaround is not needed for the proposed algorithm even though it is not completely free from unnecessary migration (which may happen due to inaccurate metrics, etc.).

The exponentially weighted averaging strategy (which was described in section 4.1.4) with an *alpha* value of 0.6, was used to capture the temporal characteristics of metrics. Note

that determining an ideal value for alpha is left for investigation in future work since this requires in-depth analysis of the relationships between the chosen alpha value, the dynamic characteristics of execution conditions, as well as the associated impact on adaptation agility and effectiveness.

For reasons similar to the experimentation concerning behaviour validation (discussed in section 5.2.1), an adaptation initiation threshold was specified such that adaptation was triggered when the availability of a particular resource was equal to or less than 80%. Again, for the same reason, the adaptation improvement score threshold for individual objects was set to 5%.

In contrast to the behaviour validation presented in sections 5.2.2 and 5.2.3, in which test applications were written so that adaptation could only be executed at specific times (in order to ensure that mobile objects have executed a pre-defined numbers of times), in this experimentation, adaptation was allowed at any point in application execution (particularly, whenever newly collected metrics reflected low resource availability). For simplicity, all mobile objects of the test applications did not maintain any state (i.e. were immutable), thus allowing migration to be performed safely during the execution of the relevant objects. Otherwise (in cases where mobile objects maintain mutable state), an advanced synchronisation mechanism is required, which would be beyond the scope of this work since it is not directly relevant to the decision making process. Note however that such configuration favours the original adaptation algorithm, since in the case where advanced synchronisation was involved, the associated overhead would heavily impact on the running application, due to the many occurrences of unnecessary migration.

The network availability of machines was controlled through the transmission of dummy data between relevant machines. CPU availability was varied by controlling the CPU consumption of an independent process: *CPU loader*, which applies a technique involving interleaving execution of CPU-bound and non-CPU-bound operations, similar to that used to vary the CPU usage intensity of objects (as described in section 5.2.2). Since both the CPU loader and the test application were executed normally, both were assigned the same scheduling priority (i.e. the default priority) and thus should have equivalent shares of the CPU.

With the exception of the last part of experiment 3 (section 5.3.4), all experiments were conducted without multi-core capability (i.e. to simulate single-core machines) because in multi-core systems, results would be heavily affected by how the underlying operating system assigned the execution of particular threads to available cores. An example scenario is a quad-core configuration, in which the CPU loader is written to launch four CPU-intensive threads in order to allow CPU load to be distributed across all four cores. The implication is that depending on how thread executions are scheduled (by the operating system), all four CPU-loading threads might get executed on three cores, leaving one core exclusively for the

test application, thereby resulting in significantly faster execution compared to the behaviour in which all CPU-loading threads execute on all cores (thus competing with the test application).

More importantly, it is difficult[3] to prevent such inconsistency in scheduling behaviour, because even slight differences (e.g. the time gap between launching the CPU loader and the test application) in the execution of the same experiment may produce significantly different results. Even more so because scheduling behaviour is hard to trace (thus hard to predict) since a particular thread may execute on different cores at different times, a phenomenon encouraged by the fact that a thread might sleep for a period of time (i.e. in order to simulate less than 100% CPU load), thereby leaving its core idle and thus available for other threads (previously running on different cores).

Preliminary experimentation results showed that the difference between multiple executions of a 2-minute experiment varied up to 30%. Consequently, despite the possibility of obtaining more consistent results through more and longer executions, the main body of this experimentation was performed without multi-core capability (allowing results to be reproduced within insignificant margins of error). The exception was the last phase of experiment 3 (section 5.3.4), in which execution using single-core and multi-core machines was compared.

5.3.2 Experiment 1: Adapting to Processor Availability

The main purpose of this experiment is to evaluate the effectiveness of adaptive application partitioning in a simple scenario involving varying processor availability. Both the proposed and the original algorithms are expected to improve the performance of the test application running on a loaded machine, with the proposed algorithm being more effective (i.e. larger performance improvement) than its counterpart due to differences in behaviour as analysed in section 5.2.

5.3.2.1 Experimental Materials

The test application used in this experiment is based on that used in the behaviour validation presented in section 5.2.2, because the base application covers sufficiently diverse objects characteristics that the differences between the original and the proposed algorithms (in terms of the quality and thus the effectiveness of decision making) can be observed. The test application contains a client object continuously accessing five objects (i.e. M1-M5) of various sizes and execution intensity as shown in Table 5-7.

[3] Although the inconsistency in scheduling behaviour may be prevented by modifying the scheduling algorithm of open-source operating systems (e.g. Linux), such a tailored implementation (which does not represent a real-world scenario) would not provide a useful benchmark.

Mobile Object	CPU Usage Intensity	In-memory Size
M1	High	Small
M2	High	Large
M3	Low	Small
M4	Low	Large
M5	Vary gradually (low to high)	Small
Where:	High = 100% CPU usage Low = 0% CPU usage	Large = 10 MB Small = 1 KB

Table 5-7. Objects with Various CPU and Memory Requirement

Mobile objects M1-M5 are executed one after another for 20 iterations. Such a configuration was chosen to allow sufficient time for: 1) necessary information/context to be collected before adaptation can be performed and 2) essential decision making behaviour to be observed. For example, in scenarios where resource availability is quite balanced across machines, migration might not occur immediately (i.e. the first time required context has been collected), because in such scenarios, decision making is more susceptible to inaccuracy in collected metrics or unintended interference (i.e. resource contention) from other processes. Another example is the unnecessary migration (e.g. ping-pong phenomenon) exhibited by the original algorithm, which is more noticeable in sufficiently long execution. Note however that, the execution duration is not so long that it greatly favours execution involving adaptation, in which case, mobile objects get to execute in more ideal execution environments (i.e. after migration) for a long period

Table 5-8 describes the machines involved in this experiment, which unlike the adopted test application, are significantly different from those used in the behaviour validation presented in section 5.2.2. This is because simulating characteristics such as runtime processor usage and memory availability (as was the case in the experiment presented section 5.2.2), is unnecessary since the aim is not to test specific decision making behaviour but rather to evaluate the effectiveness of adaptation.

Machine	Host Processor Availability (HPA)
Source	Vary Independently (0%-100%)
T1	75%
T2	75%
T3	100%
T4	85%

Table 5-8. Execution Environments with Varying CPU

As stated in Table 5-8, the CPU availability of the source machine was varied[4] (from 0% to 100% in 20% increments) in separate runs in order to facilitate more comprehensive analysis of the difference in behaviour (and thus effectiveness) between the compared adaptation algorithms. Amongst all target machines (T1-T4), the target machine T3 represents the most ideal execution environment, whereas the others were configured to represent less desirable alternatives in order to emphasize the main difference between the compared algorithms in terms of their vulnerability to unnecessary migrations (e.g. migrating objects to less powerful machines) as explained in section 5.2.2.

5.3.2.2 Analysis of Effectiveness When Adapting to Processor Availability

As depicted in Figure 5-3, without adaptation, the performance (i.e. duration in seconds) of the test application degrades exponentially as the load of the source machine increases, which is as expected, since if an application executes for 1 second at 0% load (i.e. 100% CPU availability), it takes roughly 1.25 seconds at 20% load (i.e. `100/(100-20)`), 1.67 seconds at 40%, 2.5 seconds at 60%, and so on. Note that however, in practice, the exhibited performance degradation is not as severe as the above calculation, since the CPU utilisation of the external CPU-loading process will also be compromised (as further discussed later in this section), given that both the test application and the CPU loader have the same process priority (and thus should have equivalent CPU share).

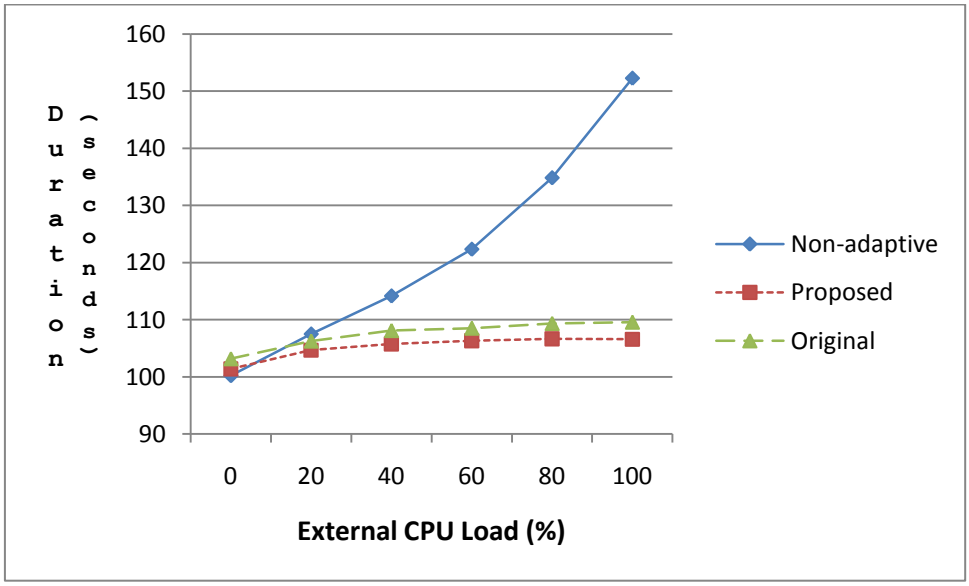

Figure 5-3. Application Performance in Various Execution Modes

In comparison, when adaptation is involved (be it using the proposed or the original algorithms), performance degradation is significantly moderated such that as load increases, ap-

[4] Note that the actual resource availability might be slightly different from the stated/intended availability due to interference from background processes.

plication performance decreases linearly. Consequently, adaptation effectiveness, which is measured as the difference in performance between non-adaptive and adaptive execution, grows exponentially to the increase in CPU load, i.e. similar to the manner in which the non-adaptive application degrades, as confirmed by the results shown in Figure 5-4. One point to note is that when the CPU load is low (i.e. 0% load), adaptive applications (using either the original or the proposed algorithm) perform slightly worse than the non-adaptive counterpart. This is because the initial placement of application objects (i.e. all located on the source machine) is already optimal, and thus not only is adaptation not beneficial, it instead causes performance degradation due to the overhead of the supporting functionality, such as metrics management and decision making.

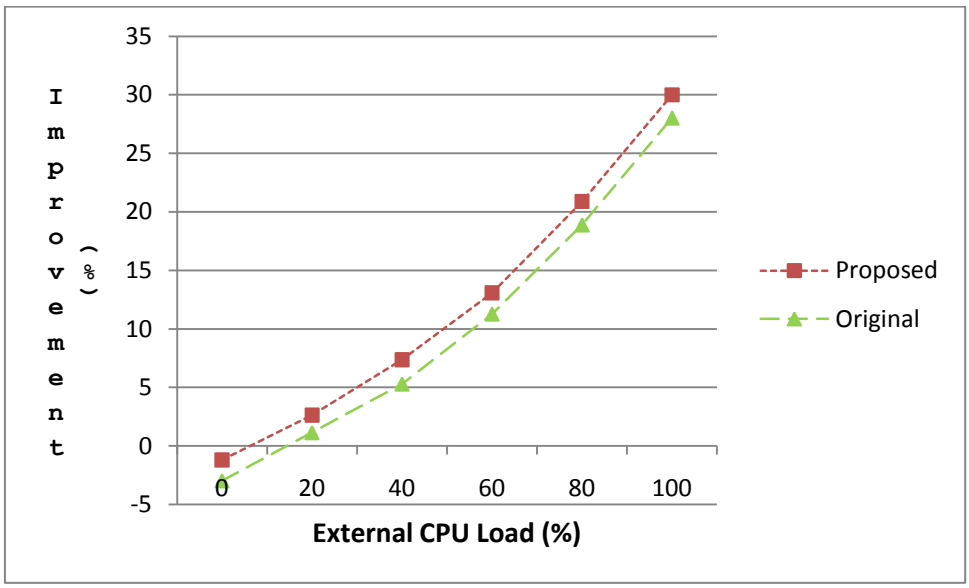

Figure 5-4. Performance Improvement Resulted from Adaptation

As shown in Figure 5-4, application performance is consistently better (i.e. between 1.5% and 2%) when the proposed algorithm is used instead of the original algorithm. This is because the latter is more susceptible to migrating objects unnecessarily due to the following reasons. Firstly, all objects are perceived as important (i.e. requiring migration) due to their long method execution, which is incorrect since long execution does not necessarily equal high CPU usage, and hence, there are not necessarily benefits from migration (to more powerful machines), as discussed in section 5.2.2. Furthermore, the original algorithm also incorrectly detects low resource availability in an adapting machine since it does not distinguish between *internal load* (contributed by the application itself or its runtime) from *external load* (contributed by other applications or processes), as confirmed in the behaviour validation presented in section 5.2.2. The consequence of such behaviour is that not only are objects incorrectly offloaded from the source machine regardless of the low external load (due to high internal load which contributes to high machine load), objects may again be migrated after

the first migration (also due to high internal load), even though there is no change in resource availability.

The performance improvement of up to 27% (as shown in Figure 5-4) is encouraging considering that unlike parallel applications, the performance of which improve relative to the number of available machines (since more machines indicate more tasks can be executed concurrently on multiple machines), sequential applications (which are the focus of this thesis) do not directly benefit from having a larger number of machines (e.g. 3 machines with 100% CPU availability do not speed up execution by 300%), although it can be argued that having more machines, especially in heterogeneous environments, provides more options to migrate a particular object with specific resource demand (e.g. high processor, low network, and low memory usage).

As mentioned, when there are other processes contending for CPU, a given process consequently gets a smaller share of CPU (i.e. the *reduced usage*) than it would otherwise (i.e. the *maximum usage*). Such behaviour also applies to the CPU-loading process, which competes with the test application, and it consequently consumes less than the configured load percentage as confirmed by the results illustrated in Figure 5-5, which show that the measured usage of the CPU loader (i.e. the *reduced usage*) never reaches 100% despite (the *maximum usage*) being configured to vary between 0 to 100%.

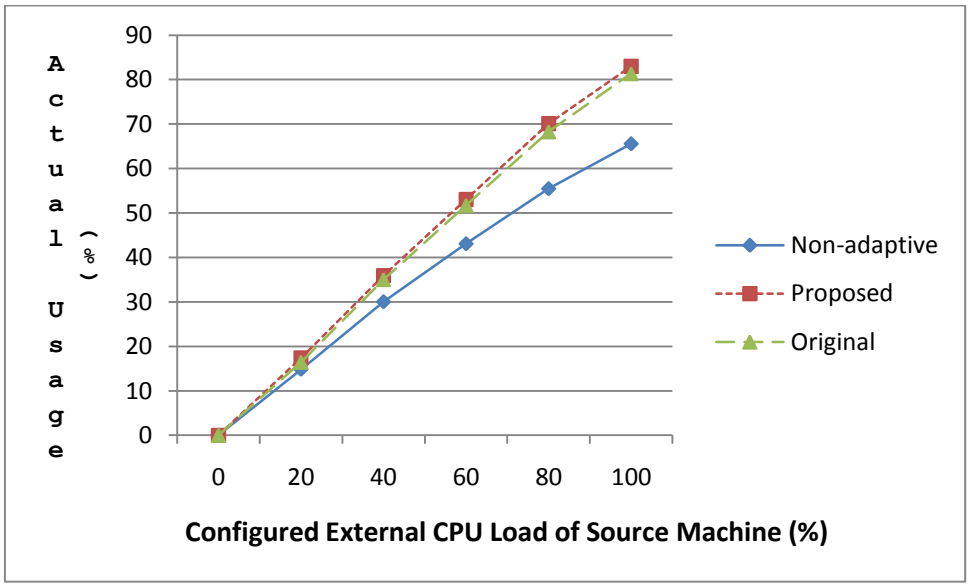

Figure 5-5. Impact of Adaptation on External CPU Usage

This phenomenon indicates a side benefit of adaptation (the main goal of which is to improve the performance of the test application), which is to relieve the load of the source machine, thus effectively providing higher CPU availability for external processes (e.g. the CPU loader) compared to the non-adaptive execution. This additional benefit, which effectively allows external processes to execute faster, is evaluated in this particular experiment through the measurement of the average CPU consumption of the CPU loader, which as depicted in

Figure 5-5, shows increases of up to 15% compared to the execution without adaptation. The graph also shows that the proposed algorithm allows slightly higher consumption by the CPU loader (i.e. between 0.5% and 1%) compared to the original algorithm due to it being more efficient (e.g. fewer migrations).

Although the *maximum usage* more accurately represents the actual load (i.e. without contention from other processes) of a machine, such information is not easily obtained[5] and thus is not used in the proposed algorithm which instead uses the *reduced usage*. As a result, the algorithm often underestimates machine load, which could result in unnecessary migrations. However, such inaccuracy is not as severe as the limitation of the original algorithm, which often significantly overestimates machine load (since both internal and external load are considered as contributing factors, as mentioned in section 5.2.2).

5.3.3 Experiment 2: Adapting to Network Availability

This experiment aims to evaluate the effectiveness of adaptation in scenarios involving complex relationships between application objects and their potential hosts (i.e. participating machines). This evaluation also concerns the relationship between the execution and interaction intensity of objects, in terms of their opposite impact on adaptation effectiveness. More specifically, unlike intensive execution, which favours adaptation (i.e. higher improvement gained from migrating objects to less loaded machines), intensive interaction incurs higher overheads for the communication between objects located on different machines.

5.3.3.1 Experimental Materials

In this experiment, the experimental materials related to the test application and its execution environment, are based on those used in the experiment presented in section 5.2.3 due to their extensive coverage of different object interaction behaviour and network characteristics. As shown in Figure 5-6, the test application consists of a stationary object S1 continuously accessing mobile objects (M1-M7) in varying manners (i.e. varying IF and SSP). The majority of the mobile objects have an outgoing relationship with another stationary object S2, with the exception of M7 which is instead coupled to M1. The mobile objects M1-M7 are executed in a loop of 10 iterations for the same reasons as those explained in the first experiment (section 5.3.2), e.g. provide sufficient time to observe interesting behaviour such as unnecessary migrations.

[5] Estimating the maximum/actual machine load would require in-depth knowledge on the thread/process scheduling behaviour of the underlying operating system.

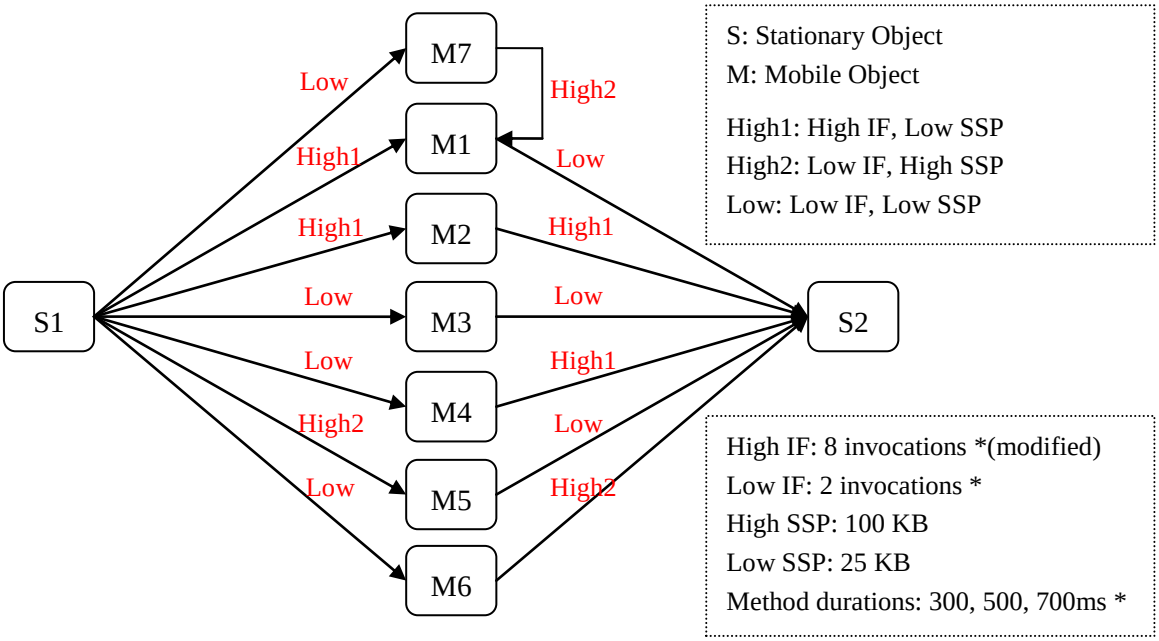

Figure 5-6. Objects with Various Interaction Characteristics

In order to facilitate the observation of the inverse relationship between execution and interaction intensity, in terms of its impact on application performance, several modifications were applied to the original test application described in section 5.2.3. First, instead of executing CPU-intensive operations for a fixed duration, the duration in which an object executes (i.e. when its method is invoked) was varied between 300, 500, and 700 milliseconds in order to have scenarios involving different execution-interaction ratios. Additionally, in order to allow a more noticeable impact when varying the aforementioned duration, object interaction intensity was moderated by reducing object/method invocations to 50% of the original.

Machine	Host Processor Availability (HPA)	Host Network Availability (HNA)
S	Vary Independently (0-100%)	Medium
T1	High	Low
T2	Medium	Medium
T3	Low	High
Where:	High = 80% of HPC Medium = 65% of HPC Low = 60% of HPC	High = 75% of HNC Medium = 50% of HNC Low = 25% of HNC

Table 5-9. Execution Environments with Varying CPU and Network Availability

The resource availability of participating machines is the same as that used in the base experiment, except that the CPU availability of the source machine is varied in separate runs, for the same reason mentioned in the first experiment (i.e. to allow more comprehensive analysis), as shown in Table 5-9.

5.3.3.2 Analysis of Effectiveness When Adapting to Network Availability

Figure 5-7, Figure 5-8, and Figure 5-9 illustrate the results of execution involving varying object execution intensity, from the lowest to the highest (i.e. 300, 500, 700 milliseconds), which confirm that the more intensive the execution (relative to the interaction), the higher the performance benefit gained from adaptation. Most notably, the second graphs (i.e. those on the right hand side) of Figure 5-7, Figure 5-8, and Figure 5-9, which present adaptation effectiveness as percentages, show that the largest performance improvement (of approximately 39%) is gained in the scenario involving the highest execution intensity (i.e. Figure 5-9). Furthermore, the second graph of Figure 5-7 (i.e. the scenario involving low execution intensity) shows that adaptation, using either the original or the proposed algorithm, causes performance degradation at 40% load, whereas Figure 5-9 (i.e. the scenario involving high execution intensity) shows that performance improves at the same load.

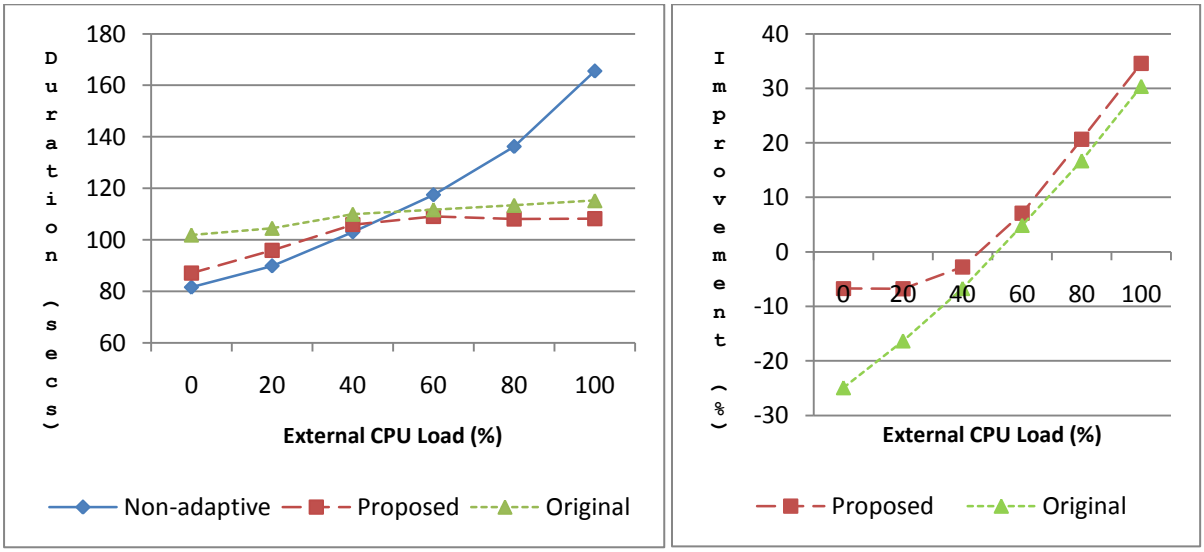

Figure 5-7. Impact of Adaptation on Application Performance (Low Execution Intensity: 300ms)

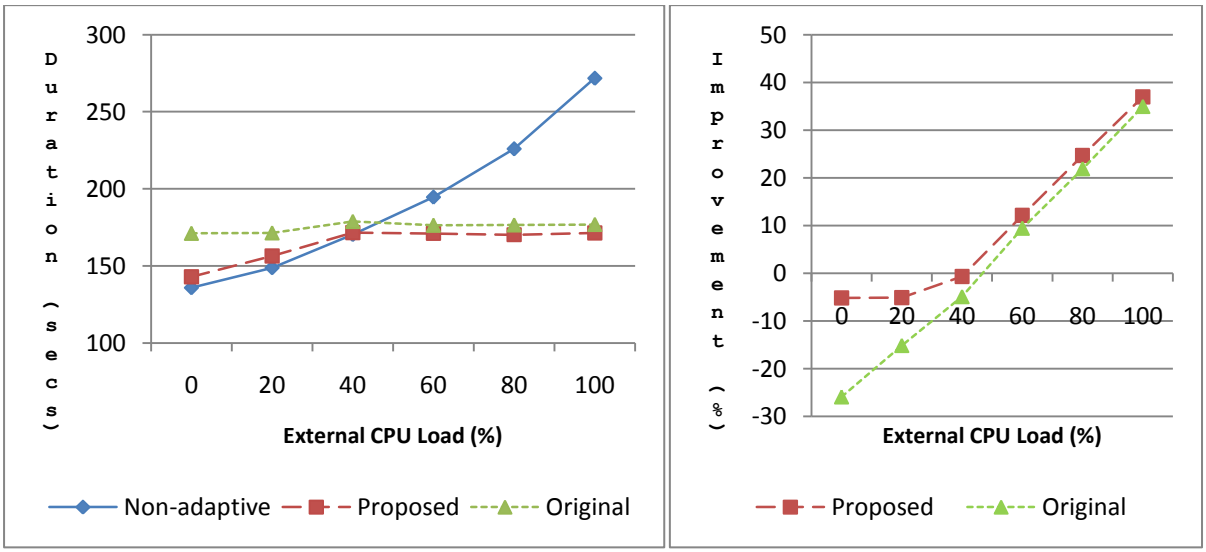

Figure 5-8. Impact of Adaptation on Performance (Medium Execution Intensity: 500ms)

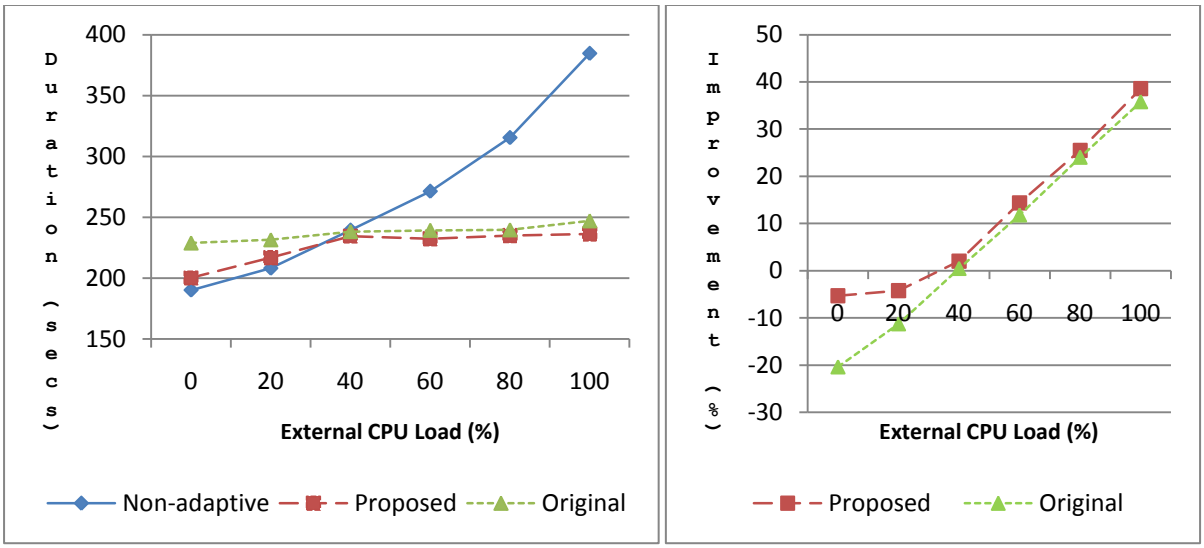

Figure 5-9. Impact of Adaptation on Performance (High Execution Intensity: 700ms)

The main cause of performance degradation (regardless of the adopted adaptation algorithm) is that the little benefit gained from adaptation due to low execution intensity and/or low source-machine load, is offset by the consequential communication overheads (due to intensive interaction between remote objects), which is exacerbated by the limited network bandwidth availability of participating machines (i.e. 25-50%). Furthermore, since the least loaded target machine (i.e. T1) has 80% CPU availability (see Table 5-9), migration is obviously not beneficial when the CPU availability of the source machine is 80% or higher (i.e. <= 20% load).

These factors are correctly interpreted by the proposed algorithm and thus, migration did not occur when the CPU load of the source machine was relatively low (e.g. 0-20%), which is why, as can be seen from Figure 5-7, Figure 5-8, and Figure 5-9, the performance degradation using the proposed algorithm (which is primarily contributed by the overheads of managing metrics) is minimal. As can be observed from the first graphs (i.e. on the left hand side) of Figure 5-7, Figure 5-8, and Figure 5-9, an interesting behaviour is shown when the CPU load is 40% or 60%, in which case, performance is slightly worse even when compared to worse cases (i.e. lower load). This is because at 40% or 60% load, unnecessary migration may occur due to the source machine S being perceived to be roughly equivalent to the target machine T1 (in terms of its suitability for hosting mobile objects), since although S is more loaded than T1 (which only has 20% load), it offers the benefit of not involving remote communication (thus not incurring overhead). Such a phenomenon may happen due to inaccuracy in collected metrics (which causes incorrect or suboptimal decisions), overestimation of machine load (as described in section 5.3.2.2), etc.

In comparison to the proposed algorithm, the original algorithm generally results in worse performance due to its tendency to migrate objects unnecessarily because of the limitations

explained in section 3.2.2.2. Furthermore, due the same limitations, object migration, which does not bring benefits when the machine load is low (e.g. 0-20%), is not prevented, thereby causing significantly worse performance degradation compared to the proposed algorithm. Note that the performance would have been worse (due to constant unnecessary migration), had a simple workaround (as described in section 5.3.1), in which decision making is prevented unless 5 seconds have elapsed since last decision making, not been applied.

5.3.4 Experiment 3: Adapting to Dynamic Execution Conditions

This experiment aims to evaluate the effectiveness of the proposed adaptation algorithm in scenarios involving dynamically changing execution conditions, including those related to the application itself (e.g. interaction behaviour) as well as its execution environment (i.e. resource availability).

5.3.4.1 Experimental Materials and Procedure

This experiment involves a total of 5 machines: one source machine (in which the application initially executes) and four target machines. Each machine executes a CPU-loading application, which plays a role similar to the CPU loader described in section 5.3.2, with the primary difference being instead of exhibiting fixed CPU and network usage, its usage, which is random (between 0-100%), changes at different points in application execution in order to simulate a dynamically changing execution environment.

Changes in resource consumption are applied every 20 seconds, a duration, which although representing a dynamic environment, is sufficiently long that the application does not constantly adapt, which would be detrimental to its performance, as discussed further in section 5.3.4.2. Given that in this experiment, the duration of application execution varies from 240 to 300 seconds (depending on factors such as dynamic versus static application behaviour, multi-core versus single-core machines as explained later), such a duration implies that adaptation is required (i.e. object placement needs to be re-adjusted) from 12-15 times.

In order to enable the reproducibility of a particular scenario, pre-defined random seeds were used to generate a sequence of random numbers, each representing a particular change in resource usage. Different seeds are used for different machines, and therefore at any one time, the participating machines may have different resource availability, although the average usage over the whole execution duration is similar for all machines (i.e. roughly 50%). As discussed further in section 5.3.4.2, two scenarios were used in order to demonstrate that despite involving randomly changing environments, some scenarios are better for adaptation (in terms of performance improvement) than others.

The test applications used in this experiment are based on the high-execution-intensity application adopted in experiment 2, because applications having lower execution intensity,

which were shown in section 5.3.3 to suffer performance degradation at 40% load, would unlikely benefit from adaptation in this experiment due to the following reasons. The average (i.e. random) CPU availability is roughly 50% in this experiment, which is lower that the highest (i.e. fixed) CPU availability in experiment 2 (which is 80%), thereby implying that less benefit may be gained from adaptation. Moreover, the continuously changing resource availability of machines represents a scenario which is worse than that applied in experiment 2, since such a scenario means that there is less time for objects to settle in an optimal placement.

This experiment is undertaken in two batches, i.e. using two different variations of the test application. The first batch uses the original form of the test application (hereinafter referred to as the *static application*), which can be considered as having "static behaviour", since it invokes (methods on) one object after another in a pre-defined sequence. The invocation sequence is repeated 10 times (thus a total of 70 invocations since the application consists of 7 mobile objects), thereby providing enough time for multiple changes in resource availability (i.e. 10-15 times depending on the actual execution duration). In the second batch, the test application is modified so that objects are invoked in a random sequence, in order to simulate "dynamic behaviour". Note that although the application (hereinafter referred to as the *dynamic application*) also involves 70 invocations, unlike the static version, mobile objects M1-M7 are not necessarily invoked the same number of times, which is why the total execution duration is different (i.e. longer than the static application) as shown in the results from section 5.3.4.2.

5.3.4.2 Analysis of Effectiveness When Adapting to Dynamic Execution Conditions

Figure 5-10 and Figure 5-11 show the duration of application execution in two execution scenarios: *bad* and *good* adaptation scenarios. The difference between the two scenarios will be outlined later in this section. Each scenario involves executions in various modes, which as shown in Figure 5-10 and Figure 5-11 (from left to right), include: 1) an execution involving a non-adaptive application running on an unloaded machine (i.e. having 100% CPU and network availability), 2) an execution of the same application on a loaded machine, 3) an adaptive execution using the proposed algorithm (on loaded machines), and 4) an adaptive execution using the original algorithm (on loaded machines). Note that although machines are loaded randomly, the same random sequences (through pre-defined seeds as mentioned in section 5.3.4.1) are used for execution in modes 2, 3, and 4.

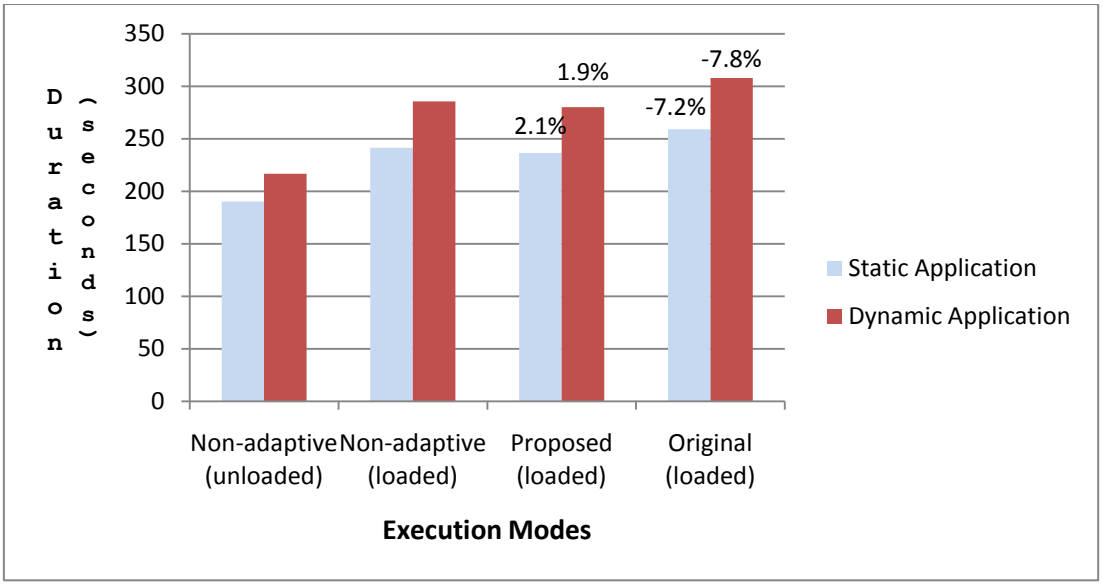

Figure 5-10. Application Performance in a Dynamic Execution Environment (Bad Scenario)

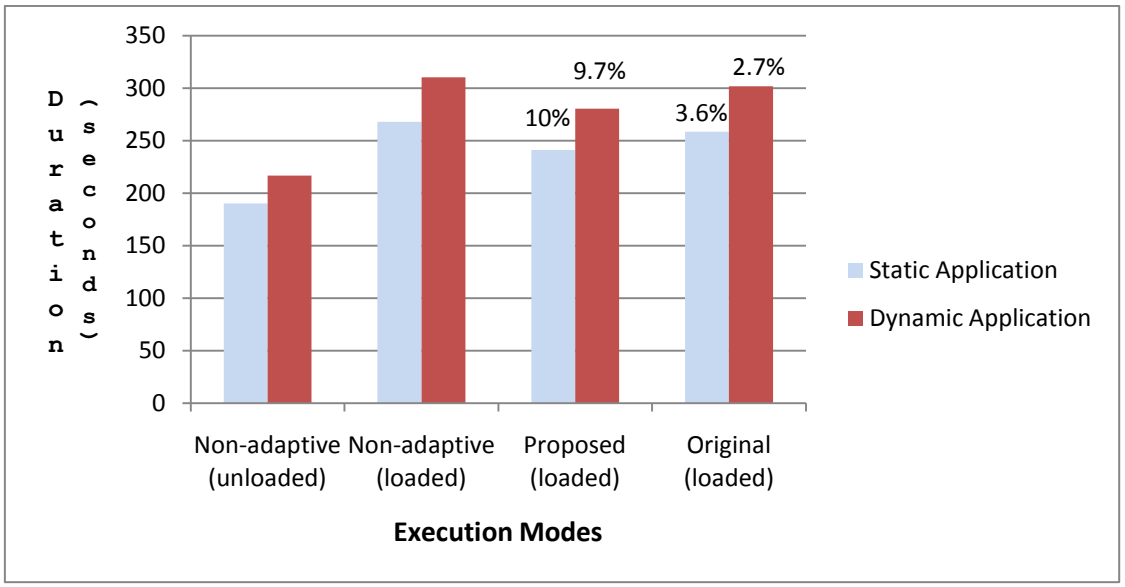

Figure 5-11. Application Performance in a Dynamic Execution Environment (Good Scenario)

The difference between the good and the bad scenarios are the seeds used to randomise the CPU load of the involved machines, which consequently impacts the effectiveness of adaptation in various ways. Firstly, in the bad scenario, the average load of the source machine is lower, a condition which does not favour adaptation since this implies less improvement from migration. Furthermore, the average load of the target machines is also slightly higher (than the good scenario), thus implying that migration (to more loaded machines) would not bring as much benefit. Last, the participating machines often have similar load at any particular time, which is not favourable to adaptation due to the limited options for migration destination (e.g. all machines are heavily loaded). In fact, such a condition could be detrimental,

since balanced machine conditions often cause unnecessary migration between machines due to factors mentioned in 5.3.3.2 (e.g. inaccuracy in collected metrics).

Consequently, compared to the good scenario, in which the proposed algorithm improves application performance by approximately 10%, adaptation in the bad scenario only brings an improvement of 2% (as can be seen from Figure 5-10 and Figure 5-11). Using the original algorithm, adaptation in fact causes performance degradation (of 7%) in the bad scenario (Figure 5-10). As such, the rest of this section focuses on the results from the good scenario since it presents more interesting results for analysis and discussion.

As can be seen from Figure 5-11, the shortest execution duration is exhibited by applications (i.e. be it *static* or *dynamic*) running in mode 1 (i.e. unloaded) which represents an ideal execution environment since there is no other process contending for resources. On the other hand, the execution involving adaptation (modes 3 and 4), which although worse off compared to the ideal case (mode 1), has better performance compared to the non-adaptive execution (mode 2).

As shown in Figure 5-11, the difference in adaptation effectiveness between execution involving the static application (i.e. 10%) and the dynamic application (i.e. 9.7%) is negligible, a result which is contrary to the general expectation that adaptation should be more effective for applications with static behaviour due to their predictability. This is because even though the static application always behaves the same way by executing objects in the same order, the interleaving execution of objects (i.e. M1-M7 execute one after another) causes inaccuracy when the adaptation algorithm assesses the importance of objects based on the recentness of their execution. For example, adaptation which is triggered immediately after the execution of a particular object (e.g. M1), would assign a higher migration priority to the object (due to its recent execution). Such behaviour is not necessarily beneficial because not only does the object (in this case M1) have short remaining duration to execute after its migration (to a more ideal machine), it would have to wait for the completion of subsequent execution (concerning other objects, e.g. M2-M7) before it can execute again. In this particular experiment, performance improvement can still be gained, because average resource load/availability changes every 20 seconds, thereby providing sufficient time for subsequent execution of M1. A more dynamic environment, wherein changes in resource availability are more frequent (i.e. less than 20 seconds), would likely result in worse performance. Such a shortcoming can be improved (in future work) by taking into account seasonal changes in metrics, which as addressed in [82], can be incorporated into the adopted metrics representation formula, i.e. exponentially weighted moving average (which is presented in section 4.1.4).

Nevertheless, the results of this experiment are encouraging since in addition to the potential improvement mentioned above, further improvement can be achieved by increasing the

number of participating nodes. This is because despite the incurred overheads, which are manageable (as will be discretely evaluated in section 5.4), it increases the probability that at any one time, there is at least one node with high resource availability, thereby enabling more effective migration (i.e. resulting in higher performance improvement compared to when all machines have low resource availability). A further improvement can be achieved by optimising the implementation of functionality involving expensive operations such as remote communication. For example, a more efficient implementation of remote invocation functionality, such as that presented in [99] and [119], can be used instead of the standard Java RMI.

Figure 5-12 shows the results of re-executing the experiment (concerning the proposed algorithm) on multi-core CPUs, which serve to demonstrate that application performance could be further improved by minimising CPU contention between operations belonging to the application itself, and the adaptation support (e.g. decision making, metrics management).

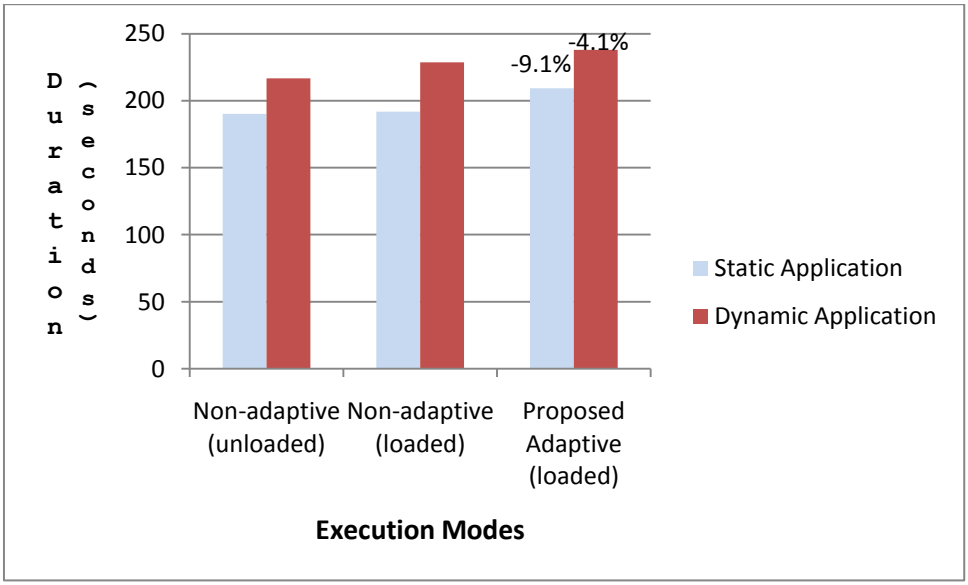

Figure 5-12. Application Performance in Multi-core Machines

However, despite the improvement in application performance compared to the single-core mode (i.e. when multi-core is disabled), the results show that adaptation actually results in performance degradation. Analysis shows that this is because increasing the usage of a multi-core CPU (i.e. loading all of its cores by the X%), does not produce the same effect as loading a single-core CPU by the same percentage. The reason being that when the CPU core on which the test application initially executes is busy, the execution may be scheduled (by the operating system) to a different core (which may be idle at the time) with minimal overhead. The exception to such behaviour is when all cores are busy (i.e. heavily loaded), which is not the case in this experiment, wherein due to the average load being roughly 50%, much of the time, the execution can be carried out immediately (on a different core) instead of having to wait for the completion of the contending execution.

As such, as can been seen from Figure 5-12, the performance of the non-adaptive application in the loaded scenario is already close to the performance in the ideal scenario where the machines are not loaded. The performance degradation exhibited by the adaptive application can be attributed to the overheads incurred by adaptation (e.g. migration, decision making, or metrics management) as well as the lack of explicit consideration for parallel execution in the proposed algorithm (since this is not the focus of this work). New metrics may be required in the future for coping with and exploiting the specific characteristic of parallel execution and multi-core CPUs.

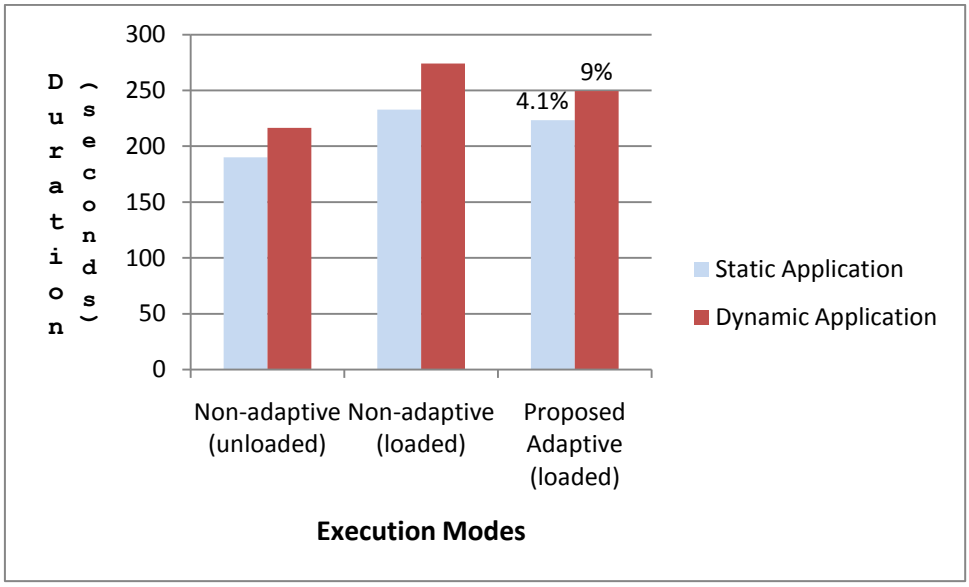

Figure 5-13. Application Performance in Multi-core Machines (Higher Load)

Nevertheless, a separate execution, which involves CPU loaders consisting of more threads (i.e. six instead of four), was conducted in order to demonstrate that applications running on heavily-loaded multi-core machines can still benefit from adaptation. As shown in Figure 5-13, such configuration, which implies an average of 75% (instead of 50%) load across machines, has more impact on application performance, and thus allows performance improvement to be gained from adaptation.

5.4 Adaptation Overhead and Scalability Evaluation

This section presents the experiments undertaken to individually evaluate the overheads of adaptation decision making (section 5.4.2) and metrics management (section 5.4.3) in terms of memory, network and processor utilisation and the associated impact on the execution duration of the adapted application. Both experiments are primarily concerned with the proposed adaptation algorithm due to its superiority (in terms of effectiveness) compared to the original algorithm, the behaviour of which is often sub-optimal due to its susceptibility to unnecessary migrations as demonstrated in sections 5.3.2 and 5.3.3. Section 5.4.1 discusses the

common experimental materials and procedure applied in both experiments (sections 5.4.2 and 5.4.3).

5.4.1 Experimental Materials and Procedure

The evaluation presented in sections 5.4.2 and 5.4.3 involved applications of various characteristics (e.g. numbers of mobile objects, number of methods per mobile objects) in order to show the scalability of the proposed adaptation solution (i.e. decision making and metrics management) as well as to serve as general guidelines for determining the overheads of adapting applications with particular characteristics (e.g. few versus many objects). Furthermore, the evaluation also helps understand how the number of available nodes (which serve as possible migration destinations) affects adaptation overheads.

Table 5-10 shows all the execution characteristics (e.g. number of objects, number of nodes) which were of concern to this evaluation due to their potential impact on various operations related to adaptation. On the other hand, other characteristics such as the number and size of parameters (passed during method invocation), were not included, due to the obvious expectation in that increasing the number and size (i.e. the complexity of object references) of invocation parameters would result in slower SSP measurement (due to serialisation overhead), but would not affect other aspects of metrics management or decision making.

Characteristic / Variation Index	1	2	3	4	5
Number of mobile objects	10	20	30	40	50
Number of methods per object	10	20	30	40	50
Number of participating nodes	10	20	30	40	50
Number of invocations	100	200	300	400	500

Table 5-10. Application Characteristics Affecting Decision Making Overhead

When evaluating the impact of varying a particular characteristic (e.g. number of objects), default fixed values were used for other characteristics, as listed in the grey column of Table 5-10 (i.e. variation index 3). The evaluation concerning a particular combination of characteristics (e.g. a scenario involving 10 objects, 30 methods, 30 nodes, and 300 invocations) was always conducted in a separate run for consistency since initial execution tends to be slower due to certain initialisations, e.g. those performed by the supporting framework (MobJeX).

Instead of aiming to represent real-world scenarios, the variation of characteristics (e.g. 10-50 mobile objects) was specified such that their impact on adaptation overheads was noticeable. Nevertheless, it is believed that the upper value of each factor represents the characteristics of large applications or complex scenarios. For example, 50, which is the largest number of mobile objects in this evaluation, represents a large-scaled application since the majority of the objects of an application are usually non-mobile (including those which form a cohesive migratable unit, e.g. non-mobile objects referenced by a mobile object). Similarly,

objects/classes containing 50 methods are generally present only in a large application, except for classes belonging to an external library (e.g. Java API), which often contain more functionality than required by the application (i.e. having many unused methods). Such classes, however, are not a concern in this evaluation because given proper configuration (during the deployment of the relevant application), unused methods would not affect adaptation decision making and metrics management (thus not incurring overheads) as addressed in section 6.3.

The number of mobile objects was varied by altering the number of iterations in which mobile objects were created, whereas different numbers of methods were obtained by authoring multiple classes having different sets of methods. Changing the number of participating nodes was simulated by running multiple host managers on a single machine due to the limited number of machines available for the experiments. Varying the number of invocations was achieved by changing the number of iterations in which a particular method of a specific mobile object was executed. The method was written to execute for approximately 50 milliseconds in order to allow sufficient time for multiple occurrences of metrics collection/delivery and decision making, before the application terminates. In its most basic form, i.e. without adaptation and using the default characteristics (i.e. 30 objects, 30 methods, 30 nodes, and 300 invocations), the application would execute for approximately 15.5 seconds.

The executed operations were 100% CPU-intensive in order to show the performance overhead of adaptation in a worst-case scenario, since application performance would be affected by the slightest contention for CPU from concurrent adaptation operations (e.g. decision making, metrics management) when running on single-core machines. For completeness, the performance overhead of adaptation was also evaluated on a multi-core system (i.e. with four cores) in order to evaluate the concurrency of the solution in terms of how performance overhead can be minimised (e.g. higher concurrency leads to better application performance since contending operations may execute on separate cores).

5.4.2 Decision Making Overheads

In addition to the general experimental procedure described in section 5.4.1, the evaluation of decision making overheads applies some specific procedure as discussed in section 5.4.2.1. The analysis of the evaluation is presented next, in section 5.4.2.2.

5.4.2.1 Experimental Procedure

In order to have a worst-case scenario, adaptation thresholds were configured in such a way that decision making was always triggered when new resource metrics (e.g. memory availability) were obtained, regardless of whether adaptation was necessary (i.e. the machine was loaded). Such configuration implied that decision making occurred every 1 second since this

was the frequency at which resource metrics (which trigger adaptation) were collected. Adaptation thresholds were also configured so that objects would always migrate (as a result of decision making), in order to further simulate a worst-case scenario, since the migration of a particular object requires adaptation scores concerning other objects to be recalculated as analysed in section 5.4.2.2. Note that the migration of objects was simulated since the aim was to evaluate decision making overheads. The simulation involved removing the model entity representing the migrated object in order to prevent the object from being processed in subsequent decision making. Such a simulation did not affect the evaluation of adaptation decision making (other than separating out migration overhead) since the evaluation only concerns a particular adapting node in a local adaptation scheme, thereby not requiring objects to be physically migrated to other nodes.

Decision making overheads were obtained by calculating the difference in measurement (e.g. of memory usage, execution duration) between the test application executing in two different modes: 1) with metrics management but without decision making, and 2) with metrics management and decision making. Such calculation produces the overheads of the decision making process, excluding the overheads incurred by automatic metrics management, which are discretely evaluated in the next section (section 5.4.3). Prior to the above calculation, the results of the measurement of resource (i.e. memory, network, and processor) usage, which occurs every 1 second, were averaged. On the other hand, execution duration (which is a measure of performance overhead) was measured once, prior to application termination.

Note that despite metrics management being the pre-requisite of decision making, the specific manner in which metrics management operations are configured (e.g. specifying criteria used for various management operations), does not affect decision making overheads and thus is not discussed in this section, but instead in the section concerning metrics management overhead (section 5.4.3.1).

5.4.2.2 Analysis of Decision Making Overheads

The results of this experiment show that despite the variation of the characteristics listed in Figure 5-14, the average memory usage of adaptation decision making is roughly constant (ranging between 160 to 190 KB), because even though the increase in the number of mobile objects, methods, etc., implies that more information needs to be processed to make adaptation decisions, this information is only temporarily stored/used (i.e. during decision making). As such, much (although not all) of this information has been discarded prior to the measurement of memory usage, since garbage collection is explicitly requested prior to each measurement.

Similarly, the average processor usage overhead of decision making is also constant, which is as expected because the relevant application is processor-bound, and thus already

exhibits maximum average processor consumption even without the additional consumption attributed to the adaptation decision making thread. On the other hand, the accumulative processor usage overhead, i.e. taking into account the duration of usage, is already implicitly measured when evaluating application performance overhead. Figure 5-14 shows the growth trend of performance overhead in relation to the increase in mobile objects, methods, nodes, and invocations. The growth is shown in terms of how much additional time (in seconds) is required to execute the application when adaptation decision making is enabled. Note that unlike other characteristics (e.g. the number of mobile objects) which are varied from 10 to 50, the number of invocations is varied from 100 to 500.

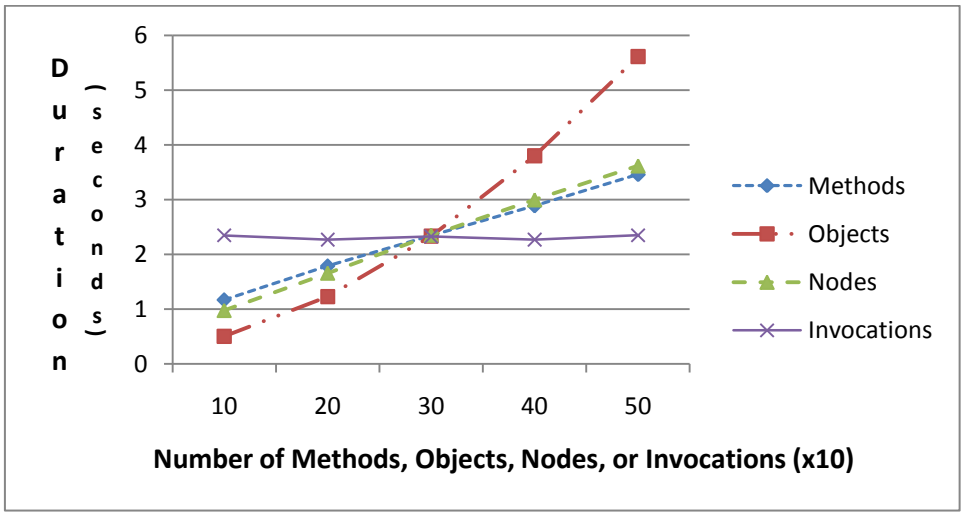

Figure 5-14. Performance Overhead of Adaptation Decision Making

As depicted in Figure 5-14, performance overhead is not affected by the number/frequency of method invocations, meaning that increasing method invocation causes longer execution duration regardless of whether adaptation decision making is enabled. On the other hand, the number of methods and nodes linearly affects performance overhead. Larger numbers of methods imply more processing required for computing the interaction intensity between objects when making adaptation decisions. Similarly, the more nodes available as possible migration destination, the more processing required to determine the suitability (i.e. adaptation score) of all object-node pairs, as explained in section 3.1.

A significantly steeper growth trend can be seen from Figure 5-14 when increasing the number of mobile objects. This is due to the behaviour of a portion of the algorithm depicted in Figure 3-1 of section 3.1, which consists of two co-existing loops that are affected by the number of mobile objects. The inner loop serves to iterate all objects to calculate the score for each object-node pair, whereas the outer loop is used to repeat the score calculation upon changes in object co-locality which are caused by the migration of a specific object. Given that the execution duration of the original non-adaptive application is approximately 15.5 seconds (as mentioned in section 5.4.1), the largest performance overhead, which was re-

corded at 3 seconds (i.e. when the number of objects equals 50), constitutes approximately 35% of the original duration, which is acceptable since the execution configuration can be considered as a worst-case scenario involving a large-scale but short-lived application running on a network of 30 nodes. Nevertheless, an improvement (to the algorithm) can be made in order to reduce the computational complexity involved, as demonstrated in ongoing work [1], which however focuses on balancing machine load as opposed to improving application performance.

In terms of network overhead, as shown in Figure 5-15, the only factor affecting network bandwidth usage is the number of nodes, since this dictates the amount of information (i.e. about individual nodes) that needs to be retrieved (via network) from a centralised context server, which runs on a separate machine. In contrast, other information, such as that related to objects and methods, is accessed locally since an adaptation engine in local adaptation only makes decisions for local objects, and therefore does not affect network bandwidth utilisation.

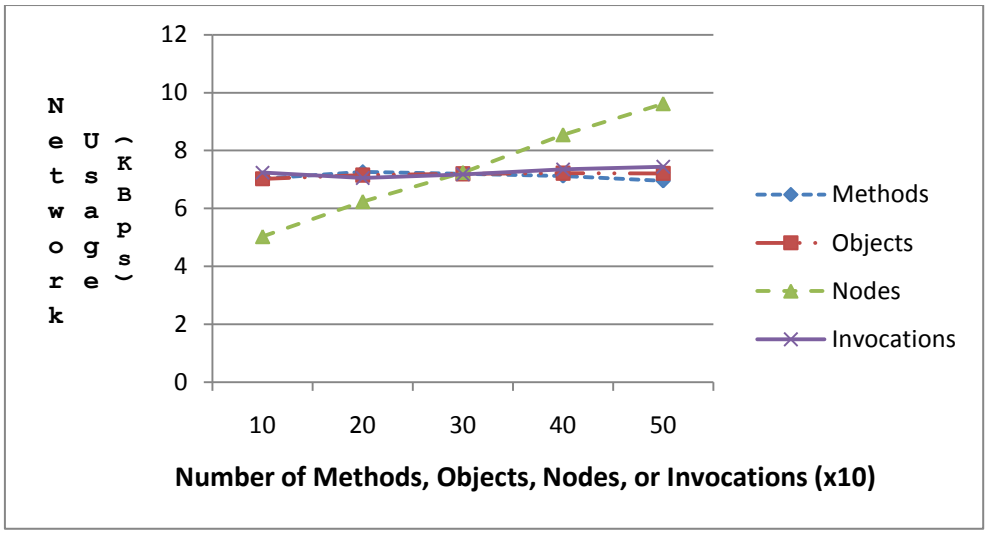

Figure 5-15. Network Usage Overhead of Decision Making

As can be seen from Figure 5-15, increasing the number of participating nodes by 10 incurs a network overhead of approximately 1 kilo byte per second (KBps). As such, since decision making is performed every second, this implies that the overhead of delivering information about a single node, is approximately 100 bytes, which is accounted for by considering the need to deliver identity information (e.g. URI) in addition to the resource metrics related to the relevant host and runtime. In practical terms, such an overhead is nominal except for execution in constrained networks (e.g. those using dialup connections). Note that this overhead is attributed to the decision making process as opposed to the metrics management tasks (which also concerns metrics delivery), because such an overhead is incurred only when decision making is enabled or involved.

The following describes the results of re-executing the same experiment on multi-core systems, which as shown in Figure 5-16, significantly reduces overhead (i.e. by roughly 95%), but does not change the trend of performance overhead (e.g. the number of mobile objects still has the largest impact). This outcome serves to demonstrate the concurrency of decision making (with regard to the adapted application), since the existence of multiple cores allows decision making tasks/threads to be executed independently with minimal impact (which still exists due to locking/synchronisation, thread scheduling, etc.) on the running application.

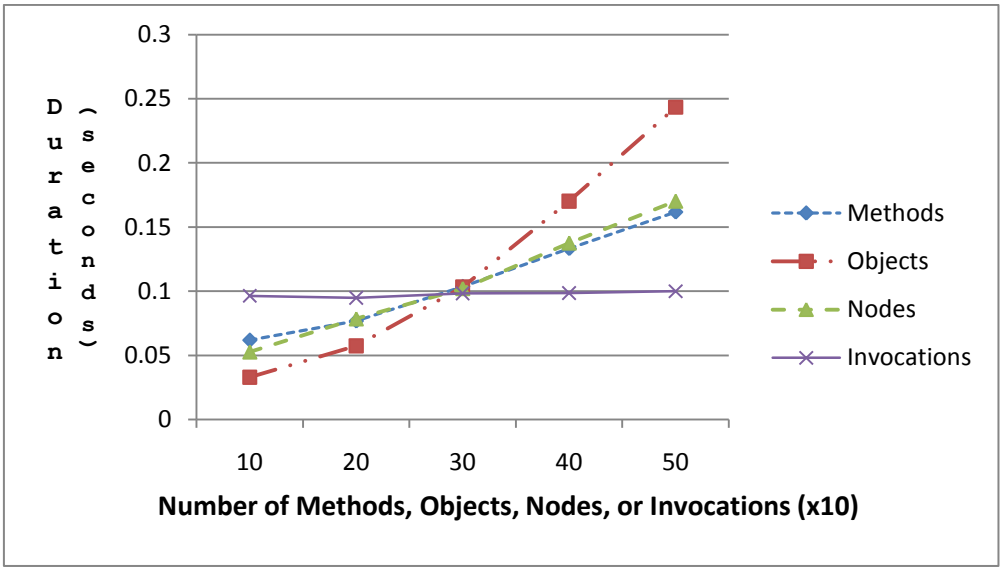

Figure 5-16. Performance Overhead of Decision Making in Multi-core Systems

Since most decision making tasks can be performed in conjunction with application execution, execution on multi-core systems also reveals the additional CPU usage required by decision making, which as shown in Figure 5-17, follows the same trend as the incurred performance overhead. Given that (as mentioned in section 5.1) a single core of the CPUs used in this experiment, can execute up to 700 million instructions per second (mips), the largest CPU usage overhead, which was recorded to be 230 million instructions per second (i.e. when the number of objects is 50), constitutes approximately 33% of the CPU capacity of the core the adaptation support is running on.

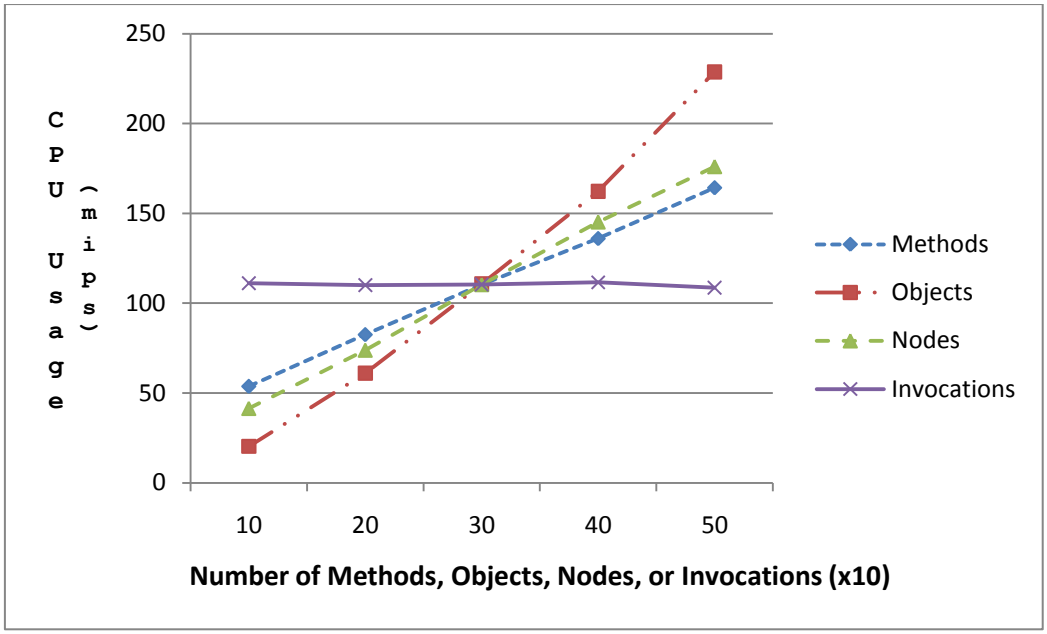

Figure 5-17. CPU Usage Overhead of Decision Making in Multi-core Systems

On a final note, according to the experiment results, the number of invocations has no impact on the overheads of decision making at all, i.e. on performance and resource usage. This factor was included in this experiment for completeness, since as discussed in section 5.4.3, it affects metrics collection/management overheads.

5.4.3 Metrics Management Overheads

This experiment aims to evaluate the overheads (i.e. performance and resource usage) of collecting and managing metrics required for adaptation, in a manner similar to that presented in section 5.4.2, with the primary difference being decision making is disabled in this experiment. Consequently, despite being related to metrics management, the delivery of remote-node metrics to the local node (i.e. the node making adaptation decisions) is not included in this evaluation, because such an activity is carried out during adaptation decision making and thus has been measured in the respective experiment (presented in section 5.4.2). In contrast, this experiment considers overheads which are incurred regardless of the execution of decision making. An example is the overhead involved in the propagation of resource metrics to a centralised context server (as addressed in section 4.2.2).

5.4.3.1 Experimental Procedure

The following describes the experimental procedure which specifically applies to this evaluation. The same test applications used in 5.4.2 were executed in two separate runs: 1) with metrics management, and 2) without metrics management. The results from the two runs were then subtracted in order to obtain the overheads (in terms of performance and resource utilisation) incurred by metrics management on the running application. For completeness, an

additional step was undertaken to measure the overheads of managing metrics on the centralised context server.

All timer-initiated measurement, such as that concerning all resource metrics and some software metrics (e.g. Object Memory Size), was performed every 1 second in order to present a worst-case scenario. For the same reason, metrics using *application-initiated* measurement were collected at every opportunity, i.e. every method invocation. Furthermore, collected metrics were always immediately propagated to the appropriate components for temporary storage (prior to delivery to adaptation engines), as described in section 4.2.2. For example, software metrics were propagated and stored in the managing runtime, whereas *runtime* resource metrics were propagated to the context server via the appropriate host manager.

5.4.3.2 Analysis of Metrics Management Overheads

The results of this experiment show that network usage is constant (approximately 4.3 kilobytes) regardless of the variation of characteristics, which include numbers of: mobile objects, methods, invocations, and nodes. One reason for this behaviour is that application-related characteristics (e.g. mobile objects, methods, and invocations) do not affect network communication since software metrics, such as Invocation Frequency (IF) and Size of Serialised Parameters (SSP), are only propagated to the containing runtime which executes in the same process as the application in which they are collected. Furthermore, the number of participating nodes does not affect the network usage of the application (and the containing runtime and host manager), since participating nodes do not communicate (i.e. exchange metrics) with each other directly, but rather through a context server, thereby affecting its network utilisation as will be discussed later in this section.

The average processor usage overhead of decision making is also constant due to the fact that the original (non-adaptive) application already consumes the maximum amount of processor time for the duration of its execution (on a single-core system), as explained in section 5.4.2. On the other hand, performance overhead, which is measured in terms of execution duration (in seconds), is shown in Figure 5-18, to increase linearly to the growing number of objects, methods, and invocations.

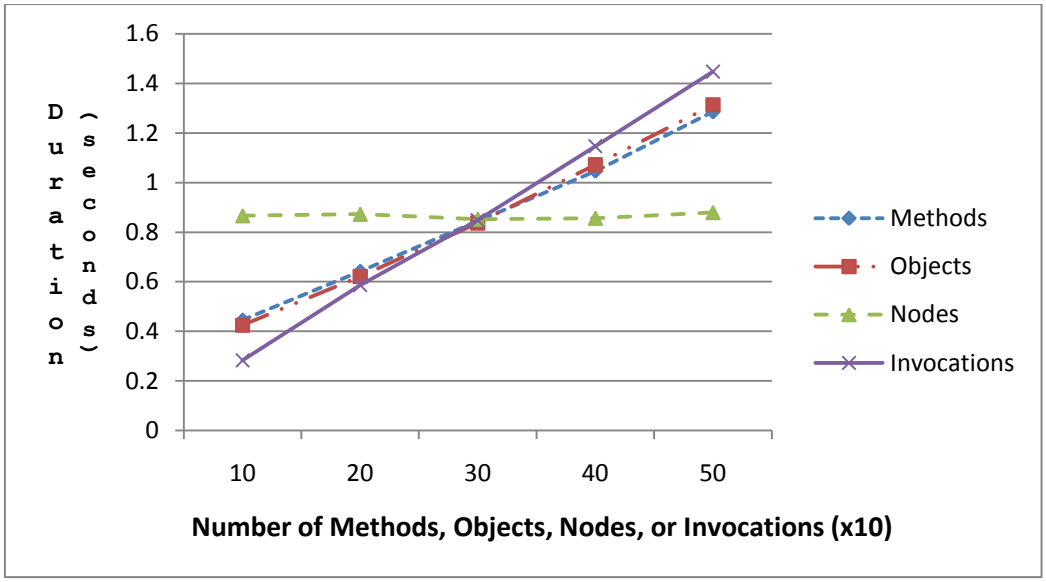

Figure 5-18. Performance Overhead of Metrics Management

Steeper overhead growth is seen when increasing method invocation, because this causes higher frequency of *application-initiated* metrics collection, which as mentioned in section 4.2.1.3, includes the potentially expensive/slow collection of the Size of Serialised Parameters (SSP) metric. On the other hand, increasing the number of methods, which consequently causes more occurrences of *timer-based* measurement (e.g. for collecting Invocation Frequency), does not cause as much overhead since the measurement is triggered at a lower frequency of every 1 second (instead of every method invocation).

In comparison, the addition of new objects incurs a similar performance overhead, since this effectively increases the overall number of methods to the same extent as varying the number of methods, e.g. adding 10 objects (i.e. 10 new objects containing 30 methods) implies an increase in the occurrence of measurement by 300. As such, in comparison, the additional overhead of collecting object-related metrics (e.g. Object Memory Size), is not as significant.

As shown in Figure 5-19, memory usage overhead is influenced by the number of objects and methods, each of which requires additional information to be maintained for the duration of application execution, and therefore increases the memory usage of the application. Examples of such information include that used for measuring object-related metrics such as Object Memory Size (OMS), and that used for measuring method-related metrics such as Processor Usage Time (PUT). Note that the lines concerning the two variation scenarios (i.e. numbers of objects and methods) are slightly adjusted for visibility purposes, but in reality there is no notable difference between them, despite the theoretical higher memory usage required for measuring object-related metrics, e.g. OMS, when increasing number of objects.

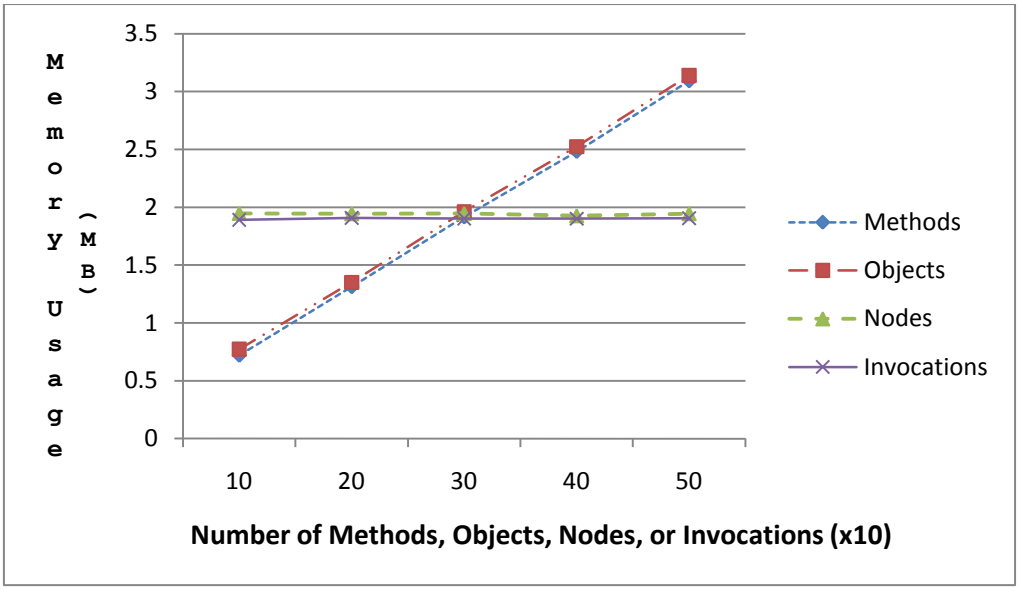

Figure 5-19. Memory Usage Overhead of Metrics Management

Although the memory usage overhead of metrics management, which is in the order of megabytes (as can be seen from Figure 5-19), may seem high, profiling shows that this is due to the prototypical nature of the implementation, which uses standard as opposed to optimised (in terms of memory usage) functionality or libraries. As an example, URIs, which are used to identify system components such as mobile objects or runtimes, can be represented in a binary instead of a human-readable form. As an implementation issue, this does not hinder the purpose of the experiment, which is primarily to show the growth trends of metrics management overhead.

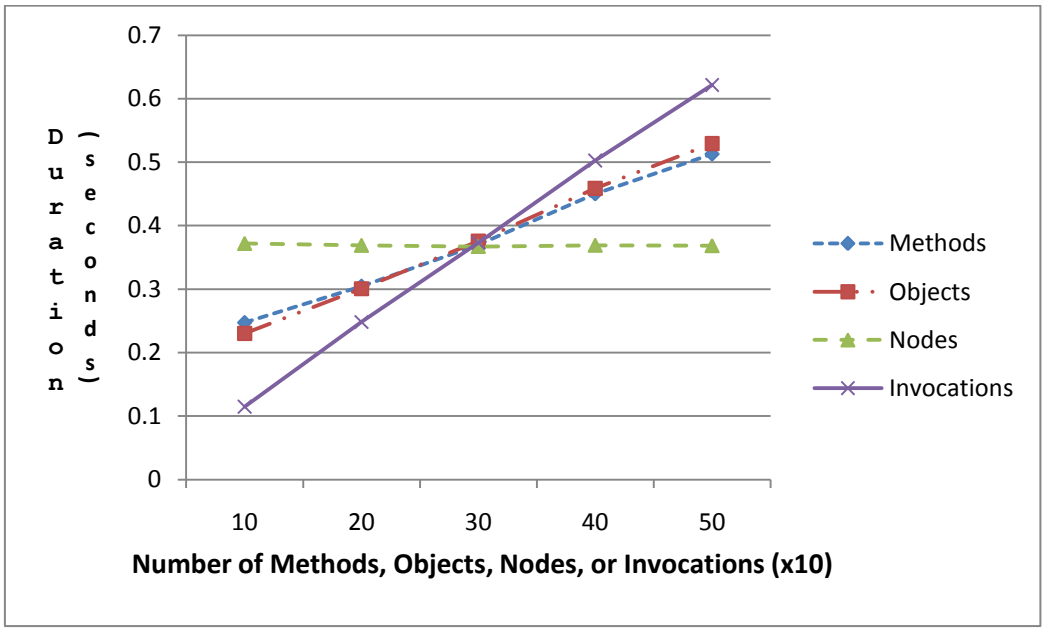

Figure 5-20. Performance Overhead of Metrics Management in a Multi-core System

Figure 5-20 shows that the use of multi-core systems reduces the performance overhead of metrics management by roughly 55%, a figure which although significantly lower compared to decision making (in which overhead is reduced by 95%), is expected considering that a substantial portion of the overhead is contributed by application-initiated measurement (introduced in section 4.1.1.1), e.g. for collecting SSP, which is inherently sequential to the execution of the running application (and thus difficult to parallelise). The supporting evidence is that in comparison to the results from the execution on a single-core system (Figure 5-18), increasing the number of invocations causes a steeper overhead growth relative to that caused by the increase in the number of objects and methods, thereby confirming that the former, which increases the frequency of application-initiated measurement, gains limited benefit from concurrency, unlike the latter, which affects the frequency of timer-based metrics (which can be collected concurrently).

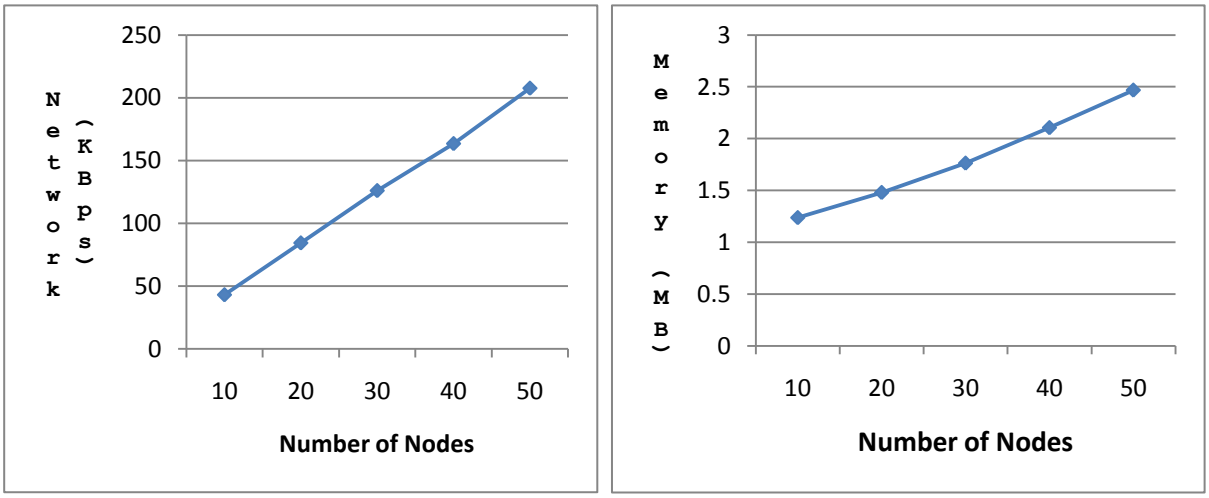

Figure 5-21. Memory and Network Usage Overheads on Context Server

As previously mentioned, despite not affecting the overhead of the running application, the number of nodes affects the resource utilization of the context server, particularly its network and memory usage, as shown in Figure 5-21. The network usage overhead, which is shown to be around 4 kilobytes per second for each node, appears high considering that only six metrics (i.e. runtime and service metrics) are propagated every second. This is caused by the utilisation of standard, un-optimised Java RMI and serialisation. Similarly, memory usage is also higher than originally expected, due to the overhead of maintaining a complete system model which consists of model entities representing participating runtimes and services (as well as their metrics) as discussed in section 4.1. However, this should not affect the scalability of the adopted solution since both network and memory usage increases linearly to the number of nodes. Note that the processor usage overhead of the context server is negligible (i.e. close to zero) and as such is not shown.

The focus of the discussion in the next chapter is the transparency of object proxy injection, which as described in section 2.5, facilitates the development of applications supporting functionality such as that concerned in this thesis, i.e. adaptive application partitioning using mobile objects.

Chapter 6. Proxy Injection

This chapter addresses the transparency of the injection of proxies into existing applications since as mentioned in section 2.6, not only do proxies play an important role in supporting adaptation via object mobility, but they have also been applied in other application domains. Such injection allows various capabilities (including those for supporting the proposed adaptation algorithm such as the collection of required metrics) to be encapsulated in proxies and be transparently injected into existing applications (in which case a higher degree of transparency implies less human intervention required in the process).

The proposed proxy solution focuses on specific code transformation for ensuring the structural and semantic compatibility of the injected proxy classes and the original/proxied class, as opposed to how the code transformation is carried out, which is the focus of Chapter 7. Structural compatibility refers to the equivalence between the structure of the proxy class and the original class in terms of type compatibility, method signatures/modifiers etc., whereas semantic compatibility ensures that the injection of proxies does not change non-application-specific semantics, such as the polymorphic behaviour of methods/invocation.

The solution improves various limitations of previous work (which are discussed in section 6.1), and thus is more transparent than existing approaches, although the exact degree of transparency was not evaluated due to the impracticality of testing such a solution against all possible scenarios/applications. Nevertheless, common transparency concerns (e.g. various inheritance scenarios, visibility modifiers) are tested in order to demonstrate the structural and semantic correctness/compatibility of the solution (as discussed in section 6.4.1).

Since a common application of proxies is to provide middleware-supported functionality, the proposed proxy approach is designed with such an application in mind and thus, where necessary, the supporting framework or middleware (e.g. MobJeX) is mentioned. Since the ideas, concepts, and solutions conceived in this work are implemented in Java, the proposed transparency solution also focuses on Java. Nevertheless, some aspects of it should also apply to languages/technologies (e.g. C#) exhibiting similar characteristics such as the lack of support for multiple-inheritance, which is generally addressed by implementing multiple interfaces (for type compatibility) and delegating method calls to other objects (to promote code reuse), as applied in the solution presented in section 6.2.1.

6.1 Transparency Issues

Sections 6.1.1-6.1.8 introduce the fundamental issues of proxy transparency, the majority of which were identified though not necessarily addressed adequately in previous work such as

[49], [178], [136]. Discussion of how existing work approached the issues is provided along with emphasis on the strengths and limitations of the approaches.

6.1.1 Proxy Inheritance

In order to maximise type compatibility between the proxy class and the original to-be-proxied class, a *proxy* class should also extend/inherit from the parent of the original class. Several aspects of proxy inheritance, such as proxying the members (i.e. fields and methods) of the parent classes and extending external classes (e.g. system/core or library classes), have yet to be adequately addressed in previous work.

The traditional proxy approach adopted in previous work, such as [147], [80], [173], uses a Java interface acting as a common interface (also known as the *domain* type) for both the *proxy* and *original* class, thereby allowing methods of the *proxy* and the *original* object to be accessed in the same manner. Ryan et al. [147] proposed a solution for allowing the proxy class to be used transparently in place of the original class.

Figure 6-1 (the left-most diagram) illustrates this approach, in which, an interface `A` is named after the original class `A` and declares the public methods of the original class. The source code of the original class is transformed into an implementation class `AImpl`[6], containing the inserted capabilities. Likewise, the generated proxy is named `AProxy`. This approach has some limitations; the major one being the inheritance hierarchy of the original class `A` is not fully reflected in the domain interface because a Java interface cannot extend a class, i.e. the parent of the original class as discussed further in section 6.2.1.3.

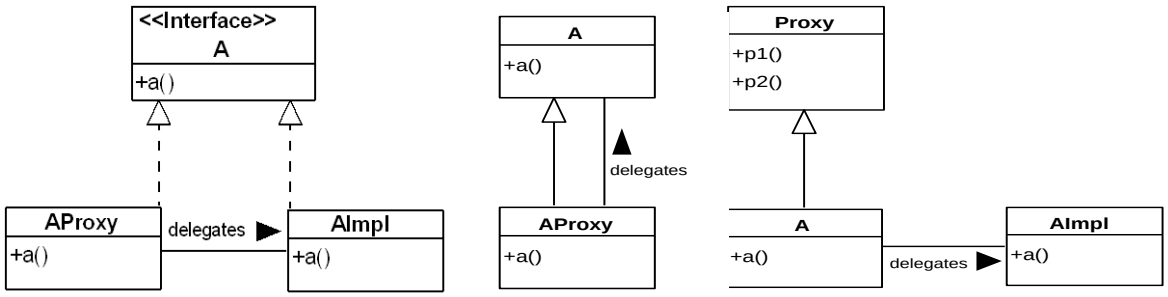

Figure 6-1. Existing Approaches (left to right): Ryan [147], Eugster [49], and JavaParty [90]/J-Orchestra [178]

Eugster [49] proposed a flexible class structure for proxies whereby the proxy class extends/inherits from the original class to maintain the type compatibility between the proxy and the original class, thereby allowing the proxy to be used wherever the original type is expected. Note that even though the proxy class `AProxy` can directly access the functionality

[6] Subsequent discussions will refer to the original class that is being, or has been, transformed as the *implementation* class as opposed to the *original* class which is the class prior to transformation.

implemented in `A` via inheritance as depicted in Figure 6-1 (the middle diagram), in practice, the proxy does not execute (i.e. completely ignores) the inherited functionality/methods, but rather it overrides the methods for the purpose of forwarding/delegating the execution to a separate instance of `A` (i.e. implementation object).

Otherwise, if `AProxy` simply uses the inherited functionality without delegating to an instance of `A`, the solution would degenerate into a *decorator design pattern* (Gamma et al., 1995). In this case, the role of the `AProxy` is to decorate (i.e. extend the functionality of) the implementation class `A`, which is not what this thesis aims to address. Such a distinction cannot be shown in a class diagram (i.e. Figure 6-1) in which instance-level relationships are absent, particularly with regard to the (inheritance and delegation) relationships between the proxy class (e.g. `AProxy`) and the implementation class (e.g. `A`). Figure 6-2 provides an instance-level illustration of the proxy approach proposed by Eugster [49].

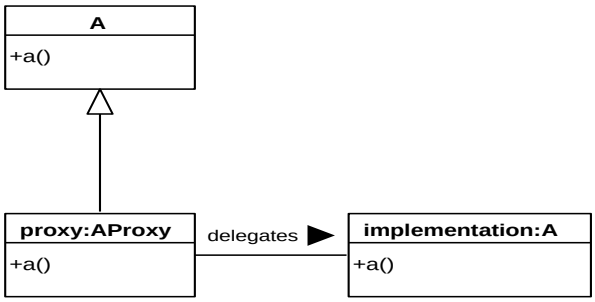

Figure 6-2. Eugster Proxy Approach [49] in Object Diagram

Although this class structure is flexible enough for accommodating the inheritance case where a proxied class `A` extends another proxied class `B` as will be discussed in section 6.2.1.2, an explicit discussion of the solution and its overhead implication was not provided in [49] and is therefore considered in section 6.2.1.2.

On the other hand, although the work on JavaParty [90] and J-Orchestra [178] addressed this inheritance case, the proxy class structure proposed in that work, which is illustrated in Figure 6-1 (right), does not accommodate the case where the parent class `B` is an external class. This is because these frameworks require the proxy class `B` to extend a *special class* (i.e. `Proxy` in the case of JavaParty and `UnicastRemoteObject` in J-Orchestra) and modifying an external class to extend the *special class* will likely affect existing dependent code, e.g. another child class of `B`, which belongs to a different application.

In order to improve proxy transparency, this thesis proposes and compares several alternative proxy class structures (in section 6.2) to address various class inheritance cases. Furthermore, other related issues such as proxying parent methods with various modifiers, e.g. final, private, etc. are also addressed (in section 6.2.2).

Subsequent sections (6.1.2-6.1.8) discuss issues which are generally applicable to all of the aforementioned proxy class structures (i.e. those adopted by Ryan [147], Eugster [49], and JavaParty [90]/J-Orchestra [178], unless explicitly specified otherwise.

6.1.2 Proxy Instantiation

In order to be able to use proxies in an ordinary Java application without explicit authoring of proxy-related code, there has to be a mechanism to transparently acquire the proxy instances. Some technologies such as Java RMI [173] and Dynamic Proxy API [170], mandate the programmer to explicitly create a proxy/stub using the provided API methods (e.g. `UnicastRemoteObject.exportObject()`). As such, any derivative of such technologies such as that proposed by Eugster [49] which was based on Dynamic Proxy, also has the same limitation.

Other technologies such as EJB 3.0 [171] provide a more transparent solution by utilising a concept called dependency injection, which prevents the programmer from having to explicitly instantiate proxies but instead provides a setter method for the proxied component (e.g. a session bean). The EJB container/middleware will then create a proxy instance and pass the instance to the setter method during the application initialisation phase. Although this approach improves transparency by removing the need for explicit instantiation, applications have to be written according to the styles/policies imposed by the framework.

In contrast, the proxy approach used by JavaParty [90] and J-Orchestra [178] provides a fully transparent means of instantiating a proxy, since the proxy is named after the original class. As such, when a client attempts to instantiate the class (e.g. using the "`new`" operator), the corresponding proxy gets instantiated instead. The instantiation of the target/original class is then done automatically (and transparently from the point of view of the client) by the proxy. Though proxy instantiation is not an issue that requires an explicit solution in JavaParty and J-Orchestra, this might not be the case in other proxy approaches (e.g. those adopted in [147] or [49]), in which case a solution is required as will be discussed in section 6.2.4.1.

6.1.3 Field Access

In Java, the client of an object interacts with the object via its fields and methods (including those that are inherited from the parent class). Methods of a target/proxied object can be invoked in a transparent manner via method call *forwarding*, wherein an invocation is forwarded from a method of the proxy to the corresponding method of the proxied object. However, this solution is not applicable for field access because unlike a method, in which the functionality can be arbitrarily defined (i.e. thus can implement the *forwarding* logic), a field

only has a fixed set of operations, i.e. read and write, thus a proxy cannot delegate to the proxied object when its field is accessed.

The solution to this issue involves generating the corresponding *field accessors* (i.e. a *setter* method for writing and a *getter* method for reading) as well as modifying the client code to invoke the appropriate *field accessors* instead of directly accessing a field. This solution has been discussed in sufficient detail in [49], and therefore section 6.2.4.2 only discusses outstanding field-related issues.

6.1.4 Reflective Operations

In addition to standard object creation using the "`new`" operator, Java provides two other ways of instantiating a class via the Java reflection API. Note that the term "*reflection*" used in this thesis should *not* be confused with the ability of an application to reflect and modify its components/behaviour accordingly, which is synonymous with application adaptation. Instead, the term "*reflection*" is used to refer to the ability to perform class/object-level operations (e.g. instantiation, method invocation) without the need to explicitly hardcode certain information (e.g. class/method names) during the development of the application.

As such, in the case of reflective instantiation, unlike the issue described in section 6.1.2, there is no reliable static/compile-time solution since the concrete type of a class accessed by reflection can only be accurately determined at runtime. Since none of the reviewed work has addressed this problem in detail, section 6.2.4.1 describes a solution to this problem so that transparency can be maintained even in the event where the proxy (or target class) is instantiated using reflection.

On the other hand, reflective method invocation does not pose a new problem since method invocation in Java is always resolved dynamically at runtime due to polymorphism, regardless of whether the invocation is done via reflection. In contrast, reflective field access introduces a problem similar to reflective instantiation, i.e. the field and the class in which the field is declared cannot always be determined at compile time as will be discussed in section 6.2.4.2.

6.1.5 Static Members

Static members (i.e. fields and methods) in Java refer to those that belong to a class rather than a specific instance/object. In traditional applications, there is only a single copy of each static member, which is shared by all instances of the same class. In a standard Java Virtual Machine (JVM), this member sharing behaviour only works within the scope of a single JVM instance/process. Consequently, this presents an issue in distributed applications, in which static fields are *not* shared between different machines since each machine loads a separate

copy of the class, therefore the value of these fields are not automatically kept consistent across machines.

J-Orchestra addressed this issue by generating an additional proxy class `SP` to manage the synchronisation of the static fields of a class. Any attempt to access a static field will be delegated to an `SP` instance to synchronise the values of the distributed static fields. Additionally, this approach also provides the flexibility of implementing extra functionality in `SP`. Since the issue of static field synchronisation has been appropriately addressed in previous work, it is not discussed any further.

6.1.6 Private Methods

Proxies are generally used to bridge inter-object communication (as mentioned in section 2.6) rather than for intra-object communication which is more appropriately addressed using an Aspect Oriented Programming (AOP) [97] technology such as AspectJ [10]. Nevertheless, private methods might need to be proxied, since even though invoking these methods is restricted to code residing in the same class, it is possible that the invocation is made by a different instance (of the same class), thus qualifying it as inter-object communication.

Proxying a private method requires the method to be exposed (e.g. making it public) so that it is accessible from the proxy. However, doing this could change the semantics of the method declaration. For example, if a class `A` has a child class `B`, exposing a private method of `A` could cause the method to be unexpectedly overridden by the child class `B` if a method with the same name exists in `B`. Such a semantic is inconsistent with that of the original class because private methods are not polymorphic (i.e. cannot be overridden) since they are not visible from the child class.

This issue was addressed by Eugster by introducing a new non-private method (called *stub method*) which simply delegates/forwards to the original private method. The stub method is given a unique name, such as being prefixed with the qualified class name (i.e. package + class name), so that it does not conflict/override a method of the parent class. Additionally, the original private method `m()` is proxied according to the following call sequence: `XProxy.m(), XImpl.X_m_stub(), XImpl.m()`, where `X_m_stub` refers to the generated stub method. However as will be discussed in section 6.2.3.2, this solution needs to be extended in order to cope with the issues introduced by proxy inheritance and parent method proxying.

6.1.7 Self Referencing

A common issue in any proxy-based approach is that passing a self-reference (i.e. using the keyword "`this`") of a proxied object to another object, will expose the proxied object, thus

undesirably allowing direct access to the object instead of via a proxy. This could break the semantic of the object interaction since the functionality supported by the proxy would not get executed in subsequent inter-object communication.

In EJB [171], this issue is addressed by requiring the developer to manually acquire the handle/proxy of a particular EJB component via the relevant API method, prior to passing/returning the proxy to other components. In contrast, the Java RMI framework automatically replaces a remote object with a proxy (RMI stub) when the object gets passed to, or returned from, a remote machine. This approach is transparent to the developer however it has the shortcoming that the automatic object substitution is only performed during remote invocation, which although appropriate for remote communication, might not be suitable for local proxy-based functionality (e.g. collecting interaction metrics)

J-Orchestra proposed that code transformation be used to substitute the "`this`" reference with the relevant proxy whenever the reference is explicitly used. Due to the minimal discussion (found in existing work) on the specific conditions that need to be satisfied for an expression to be substituted (i.e. *substitution rules*), such conditions/rules will be presented in section 6.2.3.1. Moreover, an extension to the original substitution technique, which allows self-references to be correctly passed/returned from any class in the inheritance hierarchy of the implementation class, will also be presented.

6.1.8 Identity Semantics

Cases that can cause issues related to the identity semantics of a proxy object were identified in [136] as follows: 1) checking reference equality using the "==" operator; 2) synchronising thread executions, and 3) testing an object type. As explained further in subsequent paragraphs, the first two cases only cause a problem when there are multiple proxy instances referring to a single object, since even though the proxies represent the same implementation object, in actuality they are not the same object. In mobile-object systems, which are distributed by nature, having multiple proxies of a single object is inevitable because multiple proxy instances are needed to communicate with a remote object from different machines.

The first case involves testing the equality (e.g. using the == operator) of two proxy instances, wherein the result might be inconsistent with the original application (i.e. when objects are directly compared without proxies) because the two proxy instances are always considered different despite referring to the same (proxied) object. This issue is addressed in section 6.2.4.3.

Similarly, synchronising multiple execution threads on a proxied object might not work if the synchronisation is performed on different proxy instances. However, this issue has been addressed in [178] and therefore further discussion is not necessary in this thesis.

Another identity-related issue concerns type checking of a proxied object, either *explicitly* using `getClass()` (e.g. `proxyOfA.getClass() == A.class`) or *polymorphically* using the "`instanceof`" operator. A transparent proxy approach should maintain type compatibility between the proxy class and the original class, so that both checking mechanisms return correct results, as addressed in section 6.2.4.3.

6.2 Transparency Solutions

Section 6.2.1 considers alternative proxy class structures based on their suitability in addressing the issue of proxy inheritance (introduced in section 6.1.1) as well as facilitating various adaptation capabilities. Next, the solution for addressing specific transparency issues discussed in section 6.1 as well as other issues arising from the solution, is presented, classified, and structured according to the classes that are transformed: the *proxy class* (section 6.2.2), the *original class* (section 6.2.3), and the *proxy clients* (section 6.2.4).

Issues related to the structural compatibility between the proxy class and the original/implementation class, which include class members (e.g. fields, methods), member modifiers (e.g. protected, final), and parent type hierarchy (i.e. super class and interfaces), are addressed in sections 6.2.1, 6.2.2, and 6.2.3. On the other hand, issues related to semantic compatibility (i.e. maintaining original application semantics) are addressed in sections 6.2.3 and 6.2.4.

6.2.1 Class Structuring Approaches

In supporting a proxy class hierarchy that is compatible with the original/proxied class, this thesis considers three alternative class structures, each of which is based on an existing approach (discussed in section 6.1.1). In each approach, two fundamental proxy inheritance cases are considered: a proxied class extending another proxied class and a proxied class extending an ordinary (non-proxied) class. The *first inheritance case* is used in an ideal scenario in which it is safe to transform the parent of the proxy class, thus allowing the parent class to also be proxied. On the other hand, the *second inheritance case* is generally applied when the parent class belongs to an *external* library (as opposed to the injected application) since structurally modifying such a class could affect dependent classes beyond those belonging to the injected application.

On the other hand, the scenario in which a non-proxied class extends a proxied class is not addressed in this thesis, since although it involves a proxy as well as class inheritance, it does not concern the type compatibility of a proxy class. Moreover, not only is this scenario uncommon, solutions for such a scenario are domain-specific, since the child (non-proxied)

class may either extend the (generated) parent proxy class or the (transformed) implementation class, depending on application requirements.

6.2.1.1 Alternative 1: the replace approach

The first approach improves upon the approach adopted in JavaParty [90] and J-Orchestra [178], in which the generated proxy class is named after the original class `A`, whereas the implementation class is named `AImpl`. In the case where a proxied class `B` extends another proxied class `A`, the proxy class `B` extends the parent proxy `A`, whereas the original class `BImpl` extends `AImpl` as illustrated in Figure 6-3. Consequently, the child proxy `B` automatically inherits all the functionality (e.g. methods) supported by parent proxy `A`, thus making parent methods (e.g. `a()`) accessible through the child proxy `B`.

The main difference between this proposed approach and the one adopted in JavaParty and J-Orchestra (explained in section 6.1.1), is that a proxy class does not extend a class other than the one extended by the original class, thus class type compatibility can be maintained. Furthermore, this thesis also considers the case where a proxied class extends a non-proxied class as illustrated in Figure 6-4. In this case, the proxy class `B` extends the unmodified parent class `A` for type compatibility. Additionally, class `B` should forward/delegate method calls to `BImpl` as well as `A` in order to maintain functional compatibility.

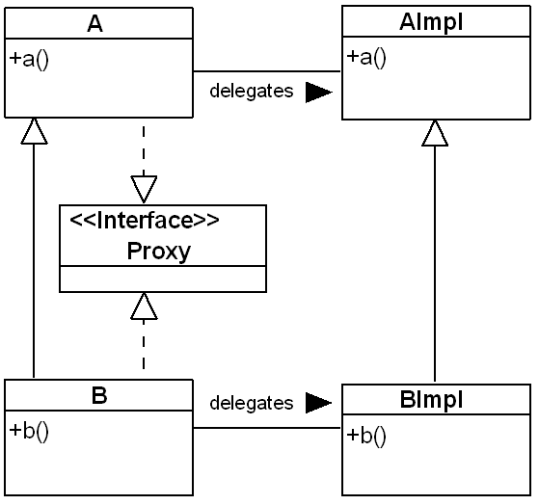

Figure 6-3. *Replace* approach: extending another proxied class A

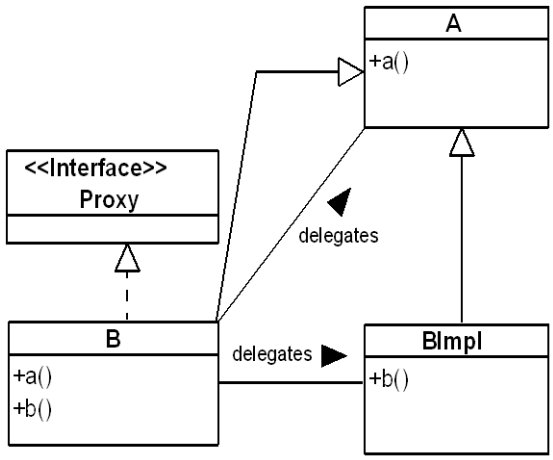

Figure 6-4. *Replace* approach: extending a non-proxied class A

In practice however, not all methods need to be proxied, since some (especially those belonging to library classes) are never invoked, and thus can be omitted from the proxy class (i.e. not forwarded/delegated). Such a case will be discussed in more detail in section 6.3, which addresses the efficiency of proxies through configurable proxy class transformation/generation. Note that proxy classes are made to implement the `Proxy` interface (as shown

in Figure 6-3 and Figure 6-4) so that they can be distinguished from the implementation objects and can be accessed in a polymorphic way by the supporting framework (e.g. MobJeX).

For clarity, subsequent discussion also refers to this proxying approach as the *replace approach*, since in this approach, the proxy class takes the name (thus the place) of the original class. The replace approach has the advantage of requiring simple code transformation when compared to the other alternatives, which will be presented in sections 6.2.1.2 and 6.2.1.3. Furthermore since the proxy class now represents (i.e. has the same name as) the original class, the semantics of type equality checking such as `proxyOfA.getClass() == A.class`, can be retained without specific code modification, which is required by the other two proxying approaches as discussed in section 6.2.4.3.

Nevertheless, this approach has a limitation in that it does not support dynamic and transparent wrapping/unwrapping of proxies (i.e. dynamically swap a proxy with the implementation object and vice versa) without breaking existing code, since there is no common type between the proxy and the implementation classes. Note that the wrapping and unwrapping functionality referred to here is dynamic, insofar as it can be performed at any point in the application execution (without breaking the client code), and as such, is more powerful than the static wrapping/unwrapping feature discussed in [178], which is only applicable at predefined places because client code has to be authored/modified accordingly.

Depending on the application/utilisation of the proxies, wrapping/unwrapping may or may not be required. For example, certain client/source objects may need to interact with a target object directly (i.e. without going through the proxy) for various reasons including: 1) improving performance and efficiency, 2) enforcing semantic correctness by preventing the execution of the proxy functionality which could undesirably modify certain states/data, and 3) addressing the issue caused by unmodifiable client code (e.g. native code) [178] [177], namely the inability to apply solutions requiring client-code modification, such as proxying field access.

Proxy wrapping/unwrapping is important in this thesis to provide flexibility in terms of the range of capabilities that can be supported or injected into an adaptive application, as will be discussed further in section 6.2.1.3. In comparison to the first (i.e. *replace*) approach, the wrapping/unwrapping capability is supported in the second (i.e. *extend*) and third (i.e. *domain*) approaches as described in sections 6.2.1.2 and 6.2.1.3.

6.2.1.2 Alternative 2: the extend approach

The second approach extends the class structure proposed by Eugster [49] to explicitly address issues in proxy inheritance. In this approach, the implementation class `A` is named after the original class `A`, while the proxy class `AProxy` extends the implementation class `A` for type compatibility. In a normal inheritance scenario (depicted in Figure 6-5), the child proxy

`BProxy` extends the implementation class `B`, which then extends the parent implementation class `A`. As such, `BProxy` is compatible with both implementation classes (i.e. `A` and `B`), whereas `AProxy` is compatible with class `A` because it extends `A`. This approach is also referred to as the *extend approach*, since type compatibility between the proxy and the original classes, is maintained by extending the original class.

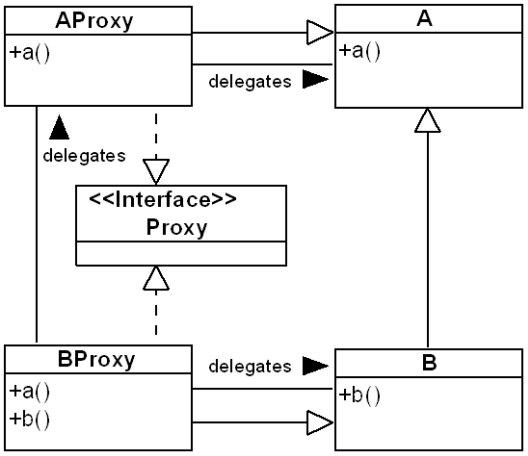

Figure 6-5. *Extend* approach: extending another proxied class A

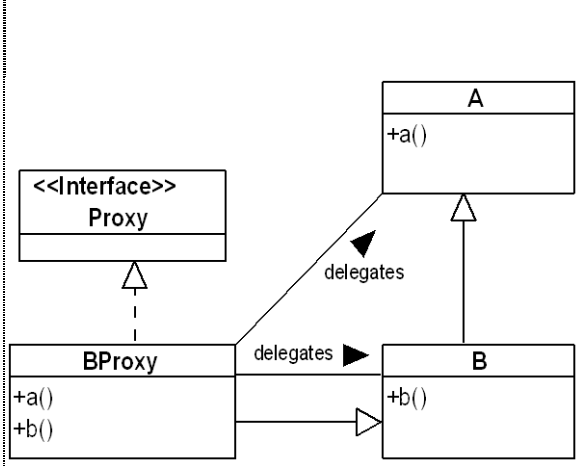

Figure 6-6. *Extend* approach: extending a non-proxied class A

Figure 6-7 illustrates (for completeness) an alternative class structure for the first inheritance case, which however has a drawback with regard to efficiency since class `B` unnecessarily inherits the functionality (i.e. methods, constructors) and data (i.e. fields) of `AProxy`, which without proper care, might also result in semantic errors. Consequently, such a class structure is less desirable in comparison to the solution depicted in Figure 6-5.

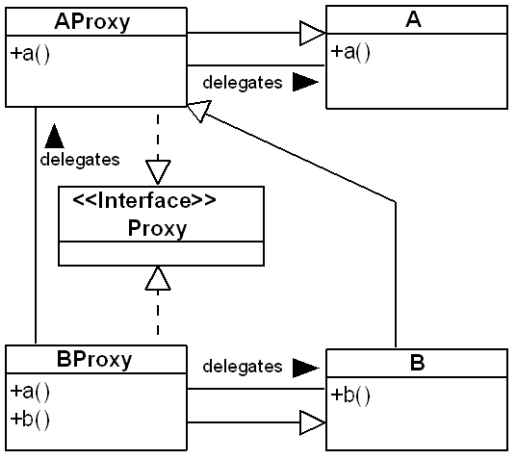

Figure 6-7. *Extend* approach: extending another proxied class A (*not preferred*)

Since in the preferred approach (i.e. Figure 6-5) the child proxy `BProxy` does not have an inheritance relationship with the parent proxy `AProxy`, it does not automatically handle (i.e.

forward/delegate) the methods of the parent class. Consequently, `BProxy`, which initially only forwards calls to methods of the child class `B`, needs to be "extended" to further allow access to methods of the parent implementation class `A`. This could be done by inserting proxy functionality for the parent methods into the child proxy, but this could result in a lot of duplication, despite effort made to minimise code generation by implementing common functionality (e.g. core metrics collection/management capability) in the supporting framework (e.g. MobJeX) instead of in the generated proxy.

As such, a delegation approach is used; wherein overridden methods are delegated to the corresponding methods of the parent proxy, i.e. `BProxy` delegates the methods of domain class `A` to `AProxy`. However, such an approach is not applicable to the second inheritance case since the parent class `A` does not have a proxy, as shown in Figure 6-6. Instead, if any of the parent methods need to be accessible from the child proxy; these methods should be inserted (with proxy functionality such as metrics management) into the child proxy `BProxy`.

As mentioned, the extend approach is more flexible than the replace approach in terms of being able to support dynamic wrapping/proxying or unwrapping/unproxying of objects. However, the drawback of this approach is that since `BProxy` extends `B`, it inherits all the fields declared in `B`, which are never used because as previously mentioned the execution of the functionality is delegated to a separate instance. Even though precautionary steps can be taken to ensure that the fields will never get initialised, they still consume a certain amount of memory.

Moreover, since the proxy automatically inherits the capabilities of the implementation class, there could be confusion or uncertainty over whether the proxy really requires/uses the capability. One such example is a proxy which indirectly implements a tagging interface[7] `java.rmi.Remote`, as a result of extending an implementation class which explicitly implements such an interface (to indicate that it supports RMI remote communication capability). In the best case, this only reduces code readability and thus complicates development and maintenance of the application/framework, but in other cases, this could incur unnecessary overheads, introduce unexpected behaviour, or even change the semantics of the original application.

Another more minor drawback is that unlike the replace approach, explicit identity checking such as `proxyOfA.getClass() == A.class` (as introduced in 6.1.8), will return a different result since `A` refers to the implementation class instead. Such an issue, which also applies to the third approach, is addressed in section 6.2.4.3. Note that this issue does not apply to polymorphic identity checking using the "`instanceof`" operator, since the proxy class (e.g.

[7] Tagging interfaces refer to interfaces which do not define any particular methods, but are used by the supporting framework (e.g. MobJeX) to identify the behaviour of classes (e.g. remote, serialisable).

`AProxy`) extends the implementation class (e.g. `A`), thus according to the inheritance rule, `AProxy` is of type `A`.

6.2.1.3 Alternative 3: the domain approach

The third approach is based upon the solution proposed by Ryan et al. [147] with the primary difference being an abstract class is used as the domain type instead of a Java interface. Such a modification allows the domain class to extend the original parent class to maintain type compatibility, which was not always possible using an interface because unlike in the first inheritance case (i.e. a proxied class `B` extends another proxied class `A)` wherein the domain interface (of the child class) `B` can extend the domain interface (of the parent class) `A` (as shown in Figure 6-8), in the second inheritance case, the parent class is not proxied and thus has no domain interface.

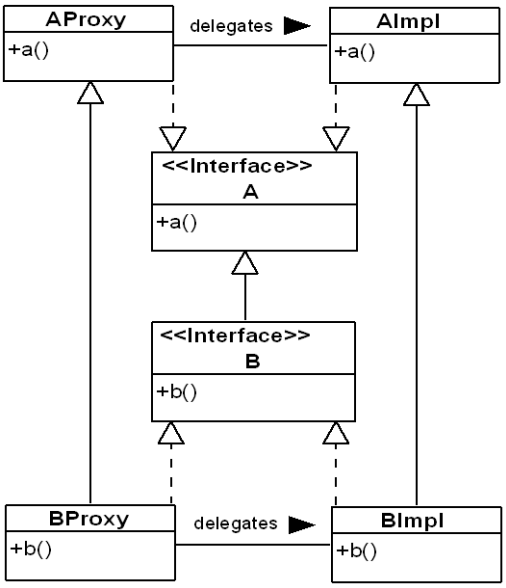

Figure 6-8. *Domain* approach: extending another proxied class A (not preferred)

On the other hand, the preferred class structuring approach, which uses an abstract class as opposed to an interface, is illustrated in Figure 6-9 and Figure 6-10. In the first inheritance case, transformation is applied to both the child class `B` and the parent class `A`, and thus there exist implementation, proxy, and domain classes for each `A` and `B`, as depicted in Figure 6-9. On the other hand, in the second inheritance scenario, the domain class (`B`) extends the original parent class (`A`) as shown in Figure 6-10. Similar to the *extend* approach, to reduce code duplication, the child proxy `BProxy` should delegate parent method invocations to `AProxy` where possible (i.e. if the parent is also proxied). Subsequent discussion refers to this approach as the *domain approach*, since a domain type is used to maintain compatibility between the proxy and the original classes.

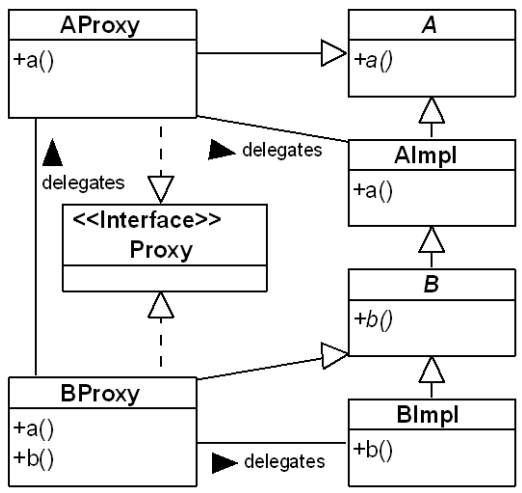

Figure 6-9. *Domain* approach: extending another proxied class A

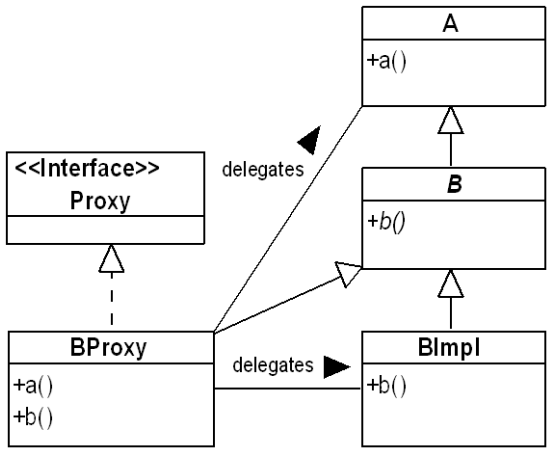

Figure 6-10. *Domain* approach: extending a non-proxied class A

In comparison to other alternatives (i.e. the replace and extend approaches), such an approach involves more complex proxy class structures. Consequently, it requires more complex code transformation, which as a result generates more code/classes as evaluated in section 6.4. Furthermore, it requires more memory than the replace approach, although not necessarily so compared to the extend approach since this depends on specific application characteristics in terms of number of fields per proxied class, etc., as evaluated in section 6.4.3. It also has the same identity checking limitation as the extend approach, which requires an additional transformation task (as addressed in section 6.2.4.3), thereby further increasing code transformation complexity.

Nevertheless, in this thesis, the *domain approach* is favoured due to its flexibility. In particular, unlike the *replace approach,* it allows wrapping and unwrapping, thus allowing a wider range of proxy functionality to be supported. One such functionality is object clustering, which allows mobile objects to be grouped as a cluster, ensuring that objects belonging to the same cluster always migrate together. Such a scenario implies that proxies and their included functionality, such as remote invocation and metrics collection capabilities, are not required for communication between objects in the same cluster as described in more detail in section Appendix A. Not only does this offer the obvious benefit of increased efficiency, but also reduces deployment complexity (thus also improving reliability) as explained in section Appendix A.

Although the larger amount of code generated in the domain approach implies higher memory overhead (since classes have to be loaded into memory before they can be used), the domain approach is also more scalable than the second alternative (i.e. the *extend approach*) as demonstrated in section 6.4.3, which shows that the memory requirement of the extend approach increases at a higher rate as the number of proxy instances increases. In addition,

the domain approach is more flexible in that although in general a proxy has to implement all the regular interfaces implemented by the original class in order to enable polymorphism, the inclusion of a tagging interface (e.g. `java.rmi.Remote`) is not necessary, allowing the decision to be made based on specific application requirements (e.g. whether a proxy should be accessible remotely). In the case of this work, although implementation objects (e.g. mobile objects) need to be accessible remotely (via a proxy), proxies do not need such functionality (thus not requiring/extending the `java.rmi.Remote` interface).

6.2.2 Parent Class Transformation

This section discusses aspects of the class structuring approaches (presented in section 6.2.1) that influence the structural properties (e.g. method visibility modifiers, constructor declarations) of a proxy class, which as a result might require changes to/transformation of the parent classes (i.e. all classes that are directly or indirectly extended by the proxy class). The majority of the described transformation is generic (i.e. applies to all class structuring approaches: *replace*, *extend*, and *domain*) unless otherwise explicitly specified. As will be revealed in this section, the modification/transformation of the parent classes is minimal and more importantly does not break dependency with other existing classes.

6.2.2.1 Parent Constructor and Initialisation

In the domain approach, the *domain* class should have a matching constructor declaration for each constructor of the original *parent* class, i.e. with exactly the same signature/arguments. These constructors are needed to forward calls from the constructors of the *implementation* class (i.e. via the "`super`" keyword) to the corresponding constructors of the original *parent*.

Similarly, an additional constructor should also be inserted into classes that are extended/inherited (directly or indirectly through other classes) by the proxy class for reasons explained in subsequent paragraphs. This requirement applies to all proxying approaches, albeit in slightly different ways. In the case of the *replace* approach, this only applies to the second inheritance case, in which the additional constructor is inserted into the parent class (e.g. class `A`) and all other classes up in the inheritance hierarchy, i.e. *indirect parent classes*, including `java.lang.Object`. In the *extend* approach, the injected classes include the implementation class and indirect parent classes, whereas in the *domain* approach, the injection involves the domain class and indirect parent classes.

Such an approach is based on a solution proposed in [49] whereby the inserted constructor receives a single argument with a unique class type, e.g. `myframework.CreationInfo` to ensure that its signature does not conflict with any of the existing constructors. The purpose of introducing a new constructor in the parent classes is to allow the proxy class to explicitly invoke the new constructor instead of relying on the default Java behaviour, which is to in-

voke an empty argument list constructor of the parent class, which might *not* exist. Even if the constructor exists, the initialisations done in the constructor could have a negative impact on performance, efficiency and transparency (e.g. unexpectedly modifying certain data/states). As such, the introduction of the new constructor also serves to prevent unnecessary or irrelevant initialisations during the construction of a proxy.

For the same reason, declaration-time field initialisations (sometimes referred to as instance initialisers) in parent classes should be prevented. This type of initialisation is similar to the initialisations done in the constructor except that they are performed earlier (i.e. prior to the constructor execution). In order to solve this issue while still maintaining the same semantics, these initialisations have to be moved to the beginning of every constructor in the class *except* for the newly inserted constructor.

6.2.2.2 Method Modifiers

This section discusses how methods with various modifiers (i.e. *final* and various visibility modifiers) can be proxied in the proposed approaches.

Any *final* (non-overridable) method of the parent class `A` that is proxied/included in the child proxy (e.g. `B` or `BProxy` depending on the adopted approach) should be made *non-final*, because proxying such a method requires the method to be overridden in the proxy. The *final* keyword is used mainly for improving code quality (e.g. maintainability, readability) and since the code transformation works on a separate copy of the code, removing this keyword during the transformation should not affect the quality of the original code. However, doing so could potentially introduce security issues, but these can be prevented if the application never uses classes from untrusted sources, which is the responsibility of the application and the supporting middleware (e.g. MobJeX), and thus is beyond the scope of this thesis.

In order to retain the accessibility semantics/constraints of proxied methods, the visibility modifiers (e.g. public, protected) of the original methods should be copied/reflected in the *implementation* class, the *proxy* class, and the domain class (if applicable). However, methods of *package-private* modifier (i.e. the default modifier in Java) are not overridable by a proxy class located in a different package, thus might require altering of the modifier to *protected*. Such a restriction only applies to methods belonging to the parent class (e.g. `A`) as opposed to the proxied class itself (e.g. `B`), since the classes might be located in packages which are different from the proxy class (in this case `BProxy`).

In addition, as mentioned in section 6.1.6, proxying a *private* method requires the insertion of a non-private stub method, which enables the call sequence: `AProxy.m()`, `AImpl.A_m_stub()`, `AImpl.m()`. In this case, the methods `AProxy.m` and `AImpl.A_m_stub` can be of package-private modifier, since it may *only* be invoked from other instances of the same class (i.e. `AImpl`), which can be placed in the same package as `AProxy`.

However, as it is, such an approach could cause a problem when a child proxy exists (e.g. `BProxy`), since `AProxy.m()` might accidentally be overridden by a method of `BProxy` that happens to have the same name. To avoid this problem, the proxy method has to be renamed, thus the call sequence becomes: `AProxy.A_m_stub(), AImpl.A_m_stub(), AImpl.m()`. This ensures that each of the parent and child classes may have a non-conflicting private method called `m()`, which can be proxied and invoked independently of each other since one is called `A_m_stub`, while the other is called `B_m_stub`. As a consequence, any code that invokes the original private method should be modified accordingly as will be discussed in section 6.2.3.2.

6.2.3 Implementation Class Transformation

This section discusses the modification of the *implementation* class required to maximise transparency in the proposed proxy approaches. Such modification only affects the internal implementation of the original class, and thus does not syntactically affect other classes. Nevertheless, since the issues addressed by this modification, which include *self-referencing* (introduced in section 6.1.7) and *proxying private methods* (introduced in section 6.2.2.2), also apply to the parent classes of an implementation class, the modification should also be performed on the parent classes. All of the modification is independent of the chosen class structuring approach.

6.2.3.1 Self-referencing

This section presents the rule for identifying the statements/expressions that need to be modified for the purpose of substituting a self-reference "`this`" of an implementation object with a proxy instance. This is so that a direct reference of the object does not get passed to other objects, which as described in section 6.1.7 could break the semantics of the proxied object interaction. Note that when substituting an occurrence of a "`this`" keyword in `AImpl`, it cannot be assumed that an instance of `AProxy` should be used as the substitution, because the "`this`" reference might refer to a child class, e.g. `BImpl`, in which case an instance `BProxy` should be used instead. As such, runtime type checking should be performed to determine the concrete type of the self-reference "`this`".

A negation is used for specifying the substitution rules, which consist of a list of cases/conditions where an occurrence of the self-reference should *not* be wrapped/substituted with a proxy. As such, during the code transformation, every occurrence of the "`this`" keyword should be checked against the conditions, and if none of them holds, the keyword gets substituted with proxy instantiation code. The conditions are as follows:

1) Method invocation, e.g `this.call()`
2) Field access, e.g. `this.var`

3) Explicit class instance method invocation, e.g. `OuterClass.this.call()`
4) Explicit class instance field access, e.g. `OuterClass.this.var`
5) Implicit `toString()` call, e.g. `println("Self: " + this)`

This negation logic serves as a safe guard where in exceptional cases (such as those when it cannot be determined whether a substitution is required), a self-reference will nonetheless be wrapped to guarantee consistent proxying behaviour. As an example of such cases, proxies are not needed when the self-reference is only passed between methods of the same object. However it is difficult to identify this case in a reliable manner without sophisticated code analysis (and complementary runtime checking) and therefore, in the absence of such code analysis support, the self-reference will be wrapped/proxied, which generally does not change the application semantic but instead introduces small performance and memory overheads. It should be noted however that it is uncommon for properly written Java code to pass a self-reference within the same object, as the reference is always directly accessible.

Additional type checking is required when applying this solution to a parent class (of a particular implementation class), since the class might be extended by other classes, which might not be proxied. Consequently, at runtime, it should be ensured that the "`this`" keyword refers to a proxied object, before the substitution can be performed.

6.2.3.2 Proxying Private Methods

As mentioned in section 6.1.6, any code that accesses a proxied private method needs to be modified to instead call the corresponding stub method. Note that since private methods are only accessible from within the declaring class, the code transformation only needs to be performed on the declaring class (i.e. implementation class). The transformation involves modifying `obj.m()` so that the "`obj`" reference is first checked whether it is a proxy, as shown in Figure 6-11.

The checking is necessary because "`obj`" could be a direct reference to the implementation object due to the unwrapping capability described in section 6.2.1.1. Furthermore, in the case where the transformation is being performed on a parent class extended by proxied and non-proxied classes, the reference "`obj`" might refer to a non-proxied object. If "`obj`" is a proxy, the relevant stub method (i.e. `A_m_stub()`) should be invoked instead, as shown in Figure 6-12.

```
// Original code in class B
B obj = ...
obj.m();

// Transformed code
(obj instanceof Proxy) ?
  obj.B_m_stub() : obj.m();
```

Figure 6-11. Private Child Method

```
// Original code in class A
B obj = ...
((A) obj).m();

// Transformed code
(obj instanceof Proxy) ?
  ((A) obj).A_m_stub() : ((A) obj).m();
```

Figure 6-12. Private Parent Method

Note that a similar solution is required for the field accessors (i.e. setter and getter methods as discussed in section 6.1.3) of a *private field*, although in this case, the insertion of a stub method is not strictly required since the field accessor itself can serve the role of a stub method, i.e. exposing the visibility of the private field and having a unique name to avoid (accidentally) overriding a parent method.

6.2.4 Client Class Transformation

This section addresses the transparency issues that are most appropriately solved by transforming *client classes*, i.e. any *application* classes (including implementation and parent classes) that potentially access another implementation class via a proxy. The involved modification only applies to the body (i.e. internal implementation not external structure) of the client class, therefore does not affect other classes. The majority of the transformation is generic, except for the proxied class instantiation solution, which as discussed in section 6.2.4.1, primarily targets the *extend approach* and the *domain approach*.

6.2.4.1 Proxied Class Instantiation

This section addresses transparent proxy instantiation as introduced in section 6.1.2, particularly targeted for the *extend approach* and the *domain approach*, since in these approaches, the proxy class uses a different name from the original class name. Consequently, the original instantiation code needs to be modified so that the corresponding *proxy* class is instantiated and returned to the *client*. The implementation class can be instantiated immediately or at a later time (e.g. when the proxy is first accessed) depending on specific application requirements.

The standard instantiation mechanism (i.e. using "`new`" operator) is addressed by traversing potential client classes and searching for the relevant instantiation code (e.g. "`new A()`"). The potential client classes are limited to those that belong to the application, since instantiating a class using the "`new`" operator requires the class name to be explicitly specified at compile time. However, this assumption does not apply to reflective instantiation (i.e. using Java reflection API), which allows dynamic resolving of class types, thus enabling instantiation to be performed by external classes, such as framework classes. As such, addressing reflective

instantiation involves traversing all classes in the execution class path to search for classes that need to be transformed, even if only a few classes require such a transformation. Such a comprehensive approach is generally not required since not only is the use of reflective instantiation rare, but it usually requires the name of the instantiated class to be explicitly specified such as via a configuration file, in which case the name of the proxy class can be specified instead (in the configuration file), thereby removing the need for code transformation.

Reflective instantiation presents another issue whereby, a static substitution/modification of code does not suffice due to the supported dynamic class resolving. As such, the solution, which relies on runtime type checking, should first replace the original instantiation code with the invocation of a newly introduced/inserted method `reflectionNew`, as shown in the example in Figure 6-13. At runtime, the method determines whether a proxy should be created based on the type of the to-be-instantiated class. Figure 6-14 shows an example specific to the domain approach, in which it is first determined whether the target class is a domain class. If it is, the relevant implementation and proxy classes will be instantiated and the proxy object will be returned to the client.

```
// Original code
java.lang.Class c =...
... c.newInstance();

// Transformed code
java.lang.Class c =...
... reflectiveNew(c);
```

Figure 6-13. Replacing Instantiation

```
Object reflectiveNew(Class c) {
  if (isDomain(c)) {
    Class ic = getImplClass(c);
    return createProxy(
        ic.newInstance());
  }
  return c.newInstance();
}
```

Figure 6-14. Proxying Created Instance

One drawback of such a solution (which requires modification of client code) is that there is no straight forward solution for the case where the client code is not modifiable, e.g. native code written in C/C++. However, it is unusual for a Java class to be instantiated from native code, since due to non-portability, native code is used sparsely and usually for low-level platform-specific operations.

6.2.4.2 Field Access of Proxied Objects

As mentioned in section 6.1.3, proxying field access has been addressed in previous work, the solution of which requires transformation of client classes. Due to the static nature of field access (i.e. non-polymorphic), the code transformation can be performed at compile time and most of the time, only the application classes need modifying since these fields are never accessed by external classes unless done via the Java reflection API.

In order to handle reflective field access, the relevant statement needs to be checked at runtime to determine whether it is accessing a field of a proxy, in which case the inserted `getFieldValue` method of the proxy is invoked instead, as depicted in Figure 6-15. The `get-`

`FieldValue` method then forwards the call to the corresponding field accessor (i.e. setter/getter method) described in section 6.1.3.

```
// Original code
Java.lang.reflect.Field f = ...
value1 = (Integer) f.get(obj);

// Transformed code
value1 = (Integer) ((obj instanceof Proxy) ?
    ((Proxy) obj).getFieldValue(f) : f.get(obj));
```

Figure 6-15. Proxying Reflective Field Access

Due to the same reason outlined in section 6.2.4.1, it is not possible to proxy a field that is accessed from unmodifiable code (e.g. native code). Nevertheless, a proxy can be unwrapped (i.e. substituted with the target object) before it is passed to the native code, in which case the field of the target object will be accessed directly.

6.2.4.3 Equality Checking

In order to address the proxy reference equality checking issue discussed in section 6.1.8, when such a statement is detected (Figure 6-16), a special method will be invoked to assist the equality checking by firstly ensuring whether the *second object* being compared is a proxy as depicted in Figure 6-17. If it is, both of the proxies being compared will be unwrapped so that the actual implementation objects can be tested for equality. In this work however, unwrapping is unnecessary, since Java RMI provides a mechanism for testing the equality of two remote objects without the need to have the objects on the same machine/JVM.

```
// Original code
if(o1 == o2) { ... }

// Transformed code
if(((o1 instanceof Proxy)?
  ((Proxy)o1).refEq(o2):(o1 == o2))){ ... }
```

Figure 6-16. Handling Reference Equality Checking

```
boolean refEq(Object o) {
  if(o instanceof Proxy) {
    return this.getImpl() ==
      ((Proxy)o).getImpl();
  }
  return false;
}
```

Figure 6-17. Testing Proxy Comparison

Note that the proxy identification (i.e. the "`instanceof`" expression) shown in Figure 6-16 only works if the compared objects are not primitive values and as such a compile-time code analysis is required to determine whether the original code should be modified.

```
// Original code
if(o.getClass() == A.class) { ... }

// Transformed code
if(((o instanceof Proxy)?
  ((Proxy)o).getOrigType() : o.getClass()) == A.class) { ... }
```

Figure 6-18. Handling Proxied Class Type Comparison

A similar approach can also be applied for checking the type of a proxied class as shown in Figure 6-18. The key to this solution is the `getOrigType` method, which should return a class reference having the same name as the original class, i.e. the proxy class in the case of the *replace* approach, the implementation class in the *extend* approach, and the domain class in the *domain* approach.

6.3 Reducing Transparency Overhead

This section addresses the configurability of the proposed proxy approaches, particularly to reduce the overheads that are introduced as a result of making them more transparent. The concept behind this configuration solution is that the application deployer will usually have knowledge of the deployed application, thereby enabling domain-specific optimisation of deployment configurations.

Firstly, the solution allows the application deployer to explicitly specify potential *client* classes. Doing this improves the efficiency of the code transformation process since the code transformation tool (provided by the supporting framework) only needs to inspect a smaller set of classes as discussed in Appendix C on the implementation of the adopted code transformation solution (in the MobJeX framework).

Moreover, the transformation also optionally allows the deployer to explicitly include/exclude individual methods of a proxied class in the generated proxy class for the purpose of reducing the storage overhead of both the transformation and the execution of the application. Non-volatile storage (e.g. disk) usage is reduced due to the smaller amount of generated code, whereas volatile storage (e.g. RAM) usage is minimised since less memory is required for supporting proxy functionality such as that for collecting method-related metrics (e.g. Invocation Frequency).

The flexibility to exclude certain methods from the proxy class is especially important if the parent class is a library class, since library classes tend to contain many more methods than will be used by a single application. As an example, proxying a class that extends the java.awt.Frame from the Java 5.0 library will generate at least 300 proxy methods, of which only a very small subset will generally be used by the application.

6.4 Evaluation

In order to facilitate the code transformation required by the solution described in sections 6.2.1-6.2.4, transformation support at both source-code and byte-code levels, has been implemented in MobJeX and a complementary transformation tool called Mobjexc. Issues related to the actual code transformation process are addressed in Chapter 7. As will be discussed in section 7.3, the proxy solution has been used to enable automatic injection of capabilities (e.g. metrics collection, location tracking) required for performing adaptive application partitioning via object mobility, thus demonstrating its transparency in terms of minimising development effort.

Since the domain approach was chosen as the primary solution for addressing proxy class inheritance in this work, the majority of the implementation focuses on this approach. Most of the transformation tasks described in sections 6.2.1-6.2.4 have been implemented, with the exception of those required for the issues of proxy reference comparison, field access, private methods, and reflective instantiation. This is because the transformation involves the complexity of requiring a certain degree of automatic code analysis, the support for which is not yet fully implemented in Mobjexc.

Arguably, the first two issues do not occur very often since firstly, in object oriented programming, logical equality checking (i.e. via the `equals` method) is generally preferred over reference comparison. Similarly, directly exposing the fields of a class is not considered good practice as it is better to use setter/getter methods which provide the flexibility to embed future functionality (e.g. value validation) into the methods. In contrast, while the last two issues are not uncommon, they are usually isolated in a specific method or class (e.g. reflective instantiation is normally handled by a factory class [60]), thus when required, manual modification can be performed on the relevant method or class.

Despite the limitations described above, the correctness, performance, and resource usage of the entire solution, including all the proposed class structures (i.e. *extend*, *replace*, and *domain*) has been evaluated as discussed in sections 6.4.1, 6.4.2, and 6.4.3. Furthermore, the limitations have been addressed by manually performing the required code modification. The experiments were undertaken using Sun JDK version 1.6.0_17 executing on a quad-core machine (i.e. Intel Q9550) running Windows XP Service Pack 3.

6.4.1 Experiment 1: Correctness

The aim of this experiment is to evaluate the correctness of the three solutions (*replace*, *extend*, and *domain*) described in section 6.2. Several test applications were written to explicitly evaluate the *structural* and *semantic* correctness (as introduced at the start of this chapter) of

the proposed proxy solutions. These applications consist of a client accessing one or more proxied classes/objects to test various scenarios, such as self referencing, field access, etc. The invocation of the inherited and overridden parent methods was also tested. Furthermore, the tests cover accessing different class members with different modifiers and each application differs in terms of the proxy inheritance hierarchy:

1) A proxied class (P) extends a non-proxied class (NP) which extends another non-proxied class (NP).
2) P extends P which extends NP
3) P extends P which extends P

Note that none of these scenarios involves a case where NP extends P, since as mentioned in section 6.2.1, such a case is not related to the inheritance hierarchy of a proxy class.

Several complex capabilities supported by the framework, such as object mobility, concurrency management, etc., were injected into the proxied classes to ensure that the insertion of the capabilities did not affect the correctness of the solutions. In addition, a separate test consisting of clients residing in different packages was conducted to more rigorously validate the visibility semantic of the proxied class.

Where possible, the tests were verified automatically using assertion statements, otherwise manual verification was done by inspecting the produced source code and tracing through the log statements printed during the execution of the test applications. The results of the tests confirmed that the transformed (i.e. proxy-based) applications exhibit the same behaviour as the original versions and proxies were indeed used where they were expected to be.

A separate test was also conducted to further verify the usability and transparency of the proposed solution. The test involved a simple GUI (Graphical User Interface) application developed using a standard graphical library, i.e. AWT (Abstract Windowing Toolkit). AWT was chosen due to the adopted class design, which involves complex inheritance and association relationships between graphical components. The test application included a frame/window containing (and thus communicating with) several user interface components (which extend `java.awt.Component`) through proxies.

Two classes were written: one extends/inherits the panel class (i.e. `java.awt.Panel`) and the other extends the button class (i.e. `java.awt.Button`), in order to address the limitation (in proxy-based solutions) in which system classes cannot be directly proxied (due to the constraint mentioned in section 6.1.1 wherein modifying such classes would affect dependent code). Next, these classes, with which the frame communicates, were transformed into proxied classes (and injected into the application). The test was considered successful since the frame including its content, was displayed, and responded to user input, in exactly the same manner as the original application (i.e. without proxies).

6.4.2 Experiment 2: Performance Overhead

Experiment 2 aims to measure the runtime performance overhead (i.e. response time) that is introduced by the proposed solution, specifically when instantiating a proxy and accessing its methods/fields. The experiment also considers operations performed using the Java reflection API, which include proxy instantiation, method invocation, and field access. The experiment includes two inheritance cases: a proxied class (P) extending a non-proxied class (NP) and a proxied class (P) extending another proxied class (P). In order to simulate a worst-case scenario in which the functionality of the original application was minimal, only basic operations such as storing and returning a single integer value, were implemented in the proxied classes. Furthermore, proxy classes were generated with no additional functionality as they simply delegated method calls to the implementation class.

All the tested operations were repeated in a 200,000 iteration loop, in order to obtain more reliable (averaged) results as well as to amplify the overhead differences between various class structuring approaches and inheritance cases. Nevertheless, the outcome shows no noticeable difference between the various scenarios, except when a method of a parent class was accessed through the child proxy. In this case, an additional overhead of roughly 0.0001 milliseconds per invocation, is introduced in the second inheritance scenario (i.e. P extends P), when either the *extend* approach or the *domain* approach is used. This is because in these approaches, method invocation needs to be forwarded from the child proxy to the parent proxy. On the other hand, such an overhead is not applicable to the replace approach because invocation forwarding is not necessary since the relevant method is inherited by the child proxy.

In comparison to the original application, in which objects are not accessed through proxies, the proxy solution (using any of the class structuring approaches) introduces 4-8 times more performance overhead for most of the tested operations. Proxy instantiation overhead was shown to be higher because in MobJeX, proxies are instantiated dynamically (i.e. without hardcoding of class names) using reflection even though the original instantiation code is static (i.e. using "`new`" operator). Such an approach provides more control and flexibility, such as the ability to unwrap a proxied object (i.e. returning a direct reference instead of a proxy) and to create multiple proxies for a single implementation object.

Upon changing the instantiation behaviour into a static approach, in which the instantiation of the proxy and the implementation classes were injected into the transformed code, the overhead was reduced to 4 times that of the original application. Another finding from this test is that field access takes roughly the same time as method invocation since it is essentially an invocation to the getter/setter of the field.

Note that although the overhead figures are relatively high (between approximately 400% and 800%), this is considered acceptable for the following reasons. Firstly, the proxy solution is no worse (i.e. no measurable difference) than a less-transparent proxy solution, as confirmed by comparing the results of the proposed approaches (i.e. replace, extend, and domain) with the well-known interface-based approach described in section 6.1.1. Secondly, the experiment was conducted using dummy classes containing minimal implementation (of insignificant processing overhead), thereby showing a worst case scenario for the resulting overhead percentages.

6.4.3 Experiment 3: Storage Overhead

This experiment aims to evaluate the impact of the adopted proxy approach on both volatile (e.g. RAM) and non-volatile (e.g. disk) storage. Non-volatile storage consumption is determined by the size of the generated classes (e.g. proxy, domain, and implementation classes), whereas volatile storage consumption is affected by the size of both the classes and runtime instances. The setup of the experiment is similar to the setup used for evaluating the performance overhead, with the exception that the client class only contains code for instantiating proxied objects since method invocation and field access are irrelevant.

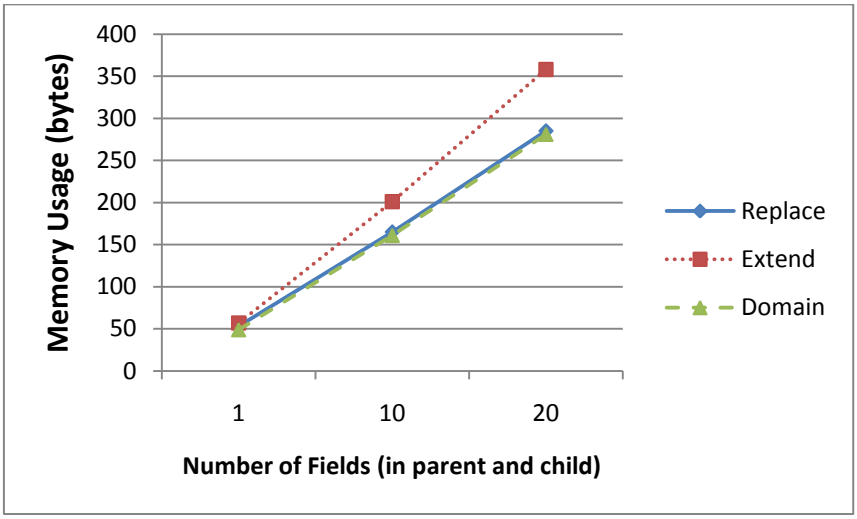

Figure 6-19. Volatile Memory Consumption in Inheritance Case 1

Figure 6-19 shows that in the first inheritance case (i.e. proxied class extends non-proxied class), both the replace approach and the domain approach consume equal amount of volatile memory (e.g. RAM). On the other hand, the extend approach consumes more memory because the proxy unnecessarily inherits the fields of the implementation class. The number of (integer) fields is varied in both the parent and the child classes between 1, 10, and 20, in order to show how it affects memory consumption.

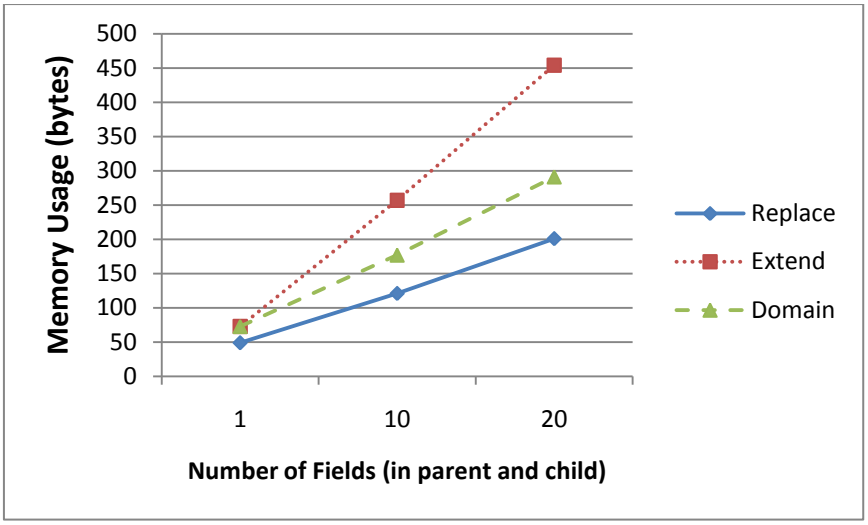

Figure 6-20. Volatile Memory Consumption in Inheritance Case 2

Figure 6-20 shows that in the second inheritance case (i.e. proxied class extends another proxied class), the replace approach has the least volatile memory consumption. On the other hand, the extend approach has the highest utilisation because the aforementioned overhead (i.e. of inheriting unnecessary fields from the implementation class) applies to both the child proxy and the parent proxy (to which the child proxy delegates).

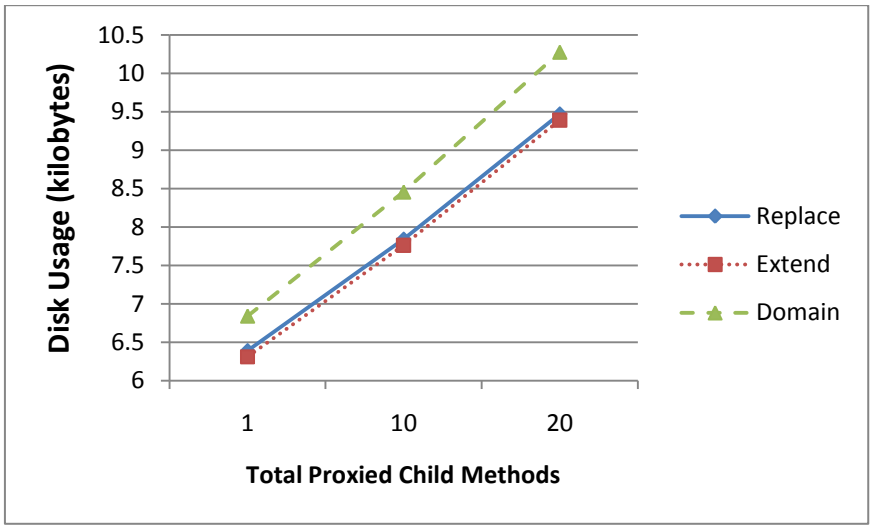

Figure 6-21. Impact of Proxied Child Methods on Non-volatile Storage Overhead

In terms of the size of the generated code artefacts, the domain approach has the largest overhead due to its more complex class structure. Figure 6-21 shows the non-volatile storage overhead of each approach (in number of bytes), which primarily depends on the number of methods that are proxied (i.e. included in the proxy class). Due to this fact, in the second inheritance case, wherein the number of proxied parent methods is varied, the replace approach has the lowest overhead as shown in Figure 6-22, because the child proxy inherits the methods of the parent proxy, and therefore does not require duplicate declaration of the same

methods (for the purpose of delegating method invocation) as is the case with the other approaches.

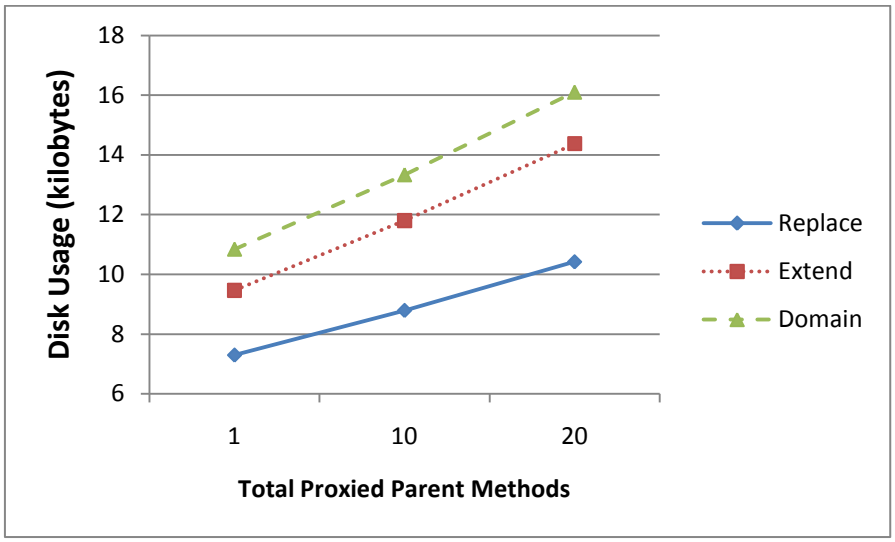

Figure 6-22. Impact of Proxied Parent Methods on Non-volatile Storage Overhead

The high non-volatile storage consumption in the domain approach (caused by the large-sized code artefacts) is not particularly a concern in this thesis since non-volatile storage is generally abundant. On the other hand, although class sizes affect volatile memory usage, classes are only loaded once (in each participating *runtime*) during the execution of an application, thus should not present scalability issues. This is in contrast to the extend approach, in which memory consumption may be a concern in large applications consisting of many proxied objects (e.g. mobile objects).

6.4.4 Summary of Evaluation

The experiment presented in 6.4.1 demonstrates the transparency of the proposed proxy solution (which includes various class structuring approaches, i.e. *replace*, *extend*, and *domain*) through the validation of structural and semantic correctness involving various transparency scenarios (e.g. inheritance, visibility modifiers). Note that the exact degree of transparency was not evaluated due to the difficulty of: 1) setting up a test which covers every possible transparency scenario (including the use of native code, which as discussed in section 6.2.4, might break transparency), and 2) establishing a metric for measuring transparency (e.g. the issue concerning native code constitutes X% of transparency).

In terms of performance, improving the transparency of a proxy-based solution does not introduce noticeable performance overhead as confirmed by the experiment presented in section 6.4.2. The results from the experiment also show that the performance difference between the different class structuring approaches (i.e. *replace*, *extend*, and *domain*) is negligible (i.e. 0.0001 milliseconds). Although the proposed proxy solution (using any of the class

structuring approaches) incurs relatively high performance overhead (i.e. 400-800%) compared to the original (non-proxied) application, such an overhead is tolerable considering the many benefits of proxies (e.g. supporting remote communication), as discussed in section 2.6.

In terms of storage, the overall requirement (be it volatile or non-volatile) of the *replace* approach is lower than the other approaches (i.e. *extend* and *domain*), as discussed in section 6.4.3. In terms of class size, which affects both non-volatile and volatile storage, the *domain* approach has the largest overhead, which however is not a major concern since non-volatile storage is not as scarce as volatile storage and since the volatile storage overhead is more scalable than that incurred by the *extend* approach, as discussed in section 6.4.3. Consequently, considering the practical benefits (e.g. flexibility in terms of supporting dynamic wrapping/unwrapping) of the domain approach over the other approaches (particularly the extend approach), as discussed in section 6.2.1.3, this thesis favours the use of the domain approach for the injection of capabilities (e.g. metrics management, object mobility) for supporting adaptive application partitioning.

The proxy injection solution proposed in this chapter is complemented by the architectural solution presented in the next chapter, which focuses on the automation of such injection using code transformation.

Chapter 7. Code Transformation

This chapter describes an architectural solution for accommodating the code transformation required by the proxy solution presented in Chapter 6, and demonstrates its applicability in terms of facilitating the injection of capabilities for supporting adaptive application partitioning via object mobility. The solution concerns requirements specific to the development of adaptive applications, which often involve manual customisation (e.g. extension, fine-tuning) of the injected adaptation functionality. Additionally, the solution addresses issues related to the transformation of distributed applications, especially with regard to the heterogeneity of target machines, since application partitioning (including object mobility) implies distribution. The aforementioned concerns and issues influence the selection of techniques (e.g. bytecode versus source-code transformation) and technologies (to facilitate the chosen transformation technique), as discussed in detail in section 7.1, which compares existing techniques/technologies and the associated benefits in terms of quality attributes such as transparency, customisability, and portability.

The presented solution primarily focuses on architectural aspects of code transformation (e.g. the software components involved and their primary roles) as opposed to specific transformation issues such as technologies and tools; however, the adoption of certain technologies and tools is also discussed in order to provide a more complete picture of the architectural solution. A more elaborate discussion of the solution (which is briefly described in section 7.2) in terms of the specific functions of individual architectural components as well as the interaction between components during the transformation of (or the injection of capabilities into) an application, is discussed in Appendix B. Furthermore, issues related to the implementation of the solution in a mobile object framework, MobJeX [147], are addressed in Appendix C.

Although the solution targets object mobility, due to its generality and flexibility, the injection of different forms of adaptation, such as that achieved by dynamically swapping an application object with another compatible object, is also supported, as discussed further in the case study presented in section 7.3. As is the case with the proxy solution, the proposed transformation solution also targets Java applications. Also note that in this work, code transformation is used to facilitate the injection of capabilities for supporting *online* adaptation (i.e. via application partitioning) into a non-adaptive application, rather than directly performing adaptation on the application code (e.g. removing certain functionality from the original application), which as discussed in section 2.1, is a process known as *offline* adaptation.

7.1 Design Decisions

Since executable units in Java are represented as classes, the transformation of a Java program is normally done by modifying individual classes. This section discusses alternative techniques and technologies for addressing the issues and challenges of class transformation in Java with an emphasis on evaluating the trade-offs between alternatives. In particular, common quality attributes such as efficiency, portability, and flexibility, are considered in addition to the aforementioned primary concerns, namely *transparency* and *customisability*.

7.1.1 Minimisation of Code Transformation

This section is concerned with the minimisation of code resulting from transformation (i.e. injection of adaptation capabilities) in order to improve efficiency, code readability, and ease of development, whilst still effectively implementing the desired behaviour. This is because although an arbitrary amount of code can be easily generated using appropriate technologies (as discussed in section 7.1.5 on generation of code fragments), it is generally not desirable to have static code duplicated in different locations since this bloats application code and thus reduces maintainability of both the transformed code as well as the transformation system itself.

Therefore, as much as possible, common code should be grouped in shared methods (e.g. as a standalone library, framework, or API) as applied in technologies, such as RMI [173], EJB [171], etc. This approach provides another advantage since not only can compilation errors in the implementation be detected at an early stage, but the behaviour and logic can also be independently tested using an automated testing framework. Note that although some technologies such as ASF+SDF [26], ensures that syntactically correct code is generated using templates (i.e. concrete syntax), the verification capability is not necessarily dependable in every possible case. For example, although it can be ensured that modifying the name of a particular class does not syntactically break the class itself, such modification may break dependent classes (which still refer to the old class name).

The proxy approach presented in section 6.2 can provide a beneficial side effect of minimising code transformation, because injected capabilities can be implemented centrally in the proxies rather than duplicated in each caller/client. Additionally, proxies allow capabilities to be injected while retaining original method signatures, thereby minimising the need to modify the relevant client code.

7.1.2 Source Code versus Byte Code

Application code transformation can be performed at different abstraction levels, including source-code and byte-code level. Although theoretically it can also be done at native-code level, the approach is not popular and is not considered in this thesis due to its lack of portability.

The main advantage of source code transformation is that the output of the transformation is human-readable, and thus facilitates manual tasks such as testing/debugging the produced code (e.g. white-box testing), implementing additional capabilities (e.g. prototyping or proof of concepts), or customising the injected functionality (e.g. fine-tuning, bug-fixing). The latter is more important in fields such as application adaptation, since both the application and the injected adaptation functionality might need to be fine-tuned according to the specific characteristics of both the target environment (software/hardware/network) and the application itself. Note that in order to accommodate the customisation of the produced source code, the framework should provide a mechanism for the application to communicate with the framework, either by specifying explicit interfaces/contracts (e.g. callback methods) to be followed by the customised application classes or by providing a set of API methods accessible from the classes.

The main drawback of source code transformation is that the source code of the application and its dependencies, is not always available and obtaining source code via decompilation (e.g. [138]) is not always appropriate due to various issues related to feasibility (e.g. lack of reliable tools) or ethicality/legality (e.g. decompiling without owner permission). Furthermore, generic decompilers do not work on protected classes (such as those that require decryption using a custom classloader or JVM), although byte-code transformation does not work in this situation either.

In contrast, whilst byte code transformation has the convenience of not requiring source code or decompilation, it is generally more cumbersome to work with, particularly for complex transformations, such as those required by the presented proxy solutions (section 6.2). Furthermore, in the absence of dedicated support tools, it is harder to debug than the source code transformation approach as evaluated in [42]. One advantage of byte code transformation is that it provides the option/flexibility to perform online transformation (i.e. at runtime).

However since this introduces overhead to the running application, which is especially of concern when the involved code transformation is as complex as that required by the proxy solution presented in section 6.2, the transformation solution presented in this thesis focuses mainly (although not solely) on source-code level transformation as discussed in section 7.1.3. The term pre-compilation is used since the transformation is performed prior to class

compilation, producing modified Java source which is then compiled to byte code using a standard Java compiler.

7.1.3 Offline versus Online

This section discusses two transformation approaches: *offline transformation*, which refers to transformation performed prior to application execution; and *online transformation*, which involves transformation performed during application execution. Offline transformation is traditionally performed at *pre-compile* time (i.e. prior to class compilation), whereas online transformation is performed at *class-load* time (i.e. when a class gets loaded by the JVM). Another offline approach known as *post-deploy-time* transformation, which refers to transformation performed after the deployment of an application but prior to its execution, is also considered in this thesis. Such an approach addresses certain limitations of pre-compile-time and class-load-time transformation as outlined in subsequent discussions which compare and contrast the different approaches.

One disadvantage of offline transformation, which applies to both pre-compile-time and post-deploy-time transformation, is that it is not adequate for certain applications of code transformation such as that described in [5] for optimising object serialisation, which should be performed at runtime since the transformation requires information that can only be obtained during the execution of the application, such as which objects are referenced by the fields of a particular object. However, such a constraint is not relevant in this work since the information required for injecting the concerned capabilities (e.g. metrics management) can be acquired statically, i.e. via static code analysis, configuration, etc.

In the pre-compilation approach, transformation is performed only as many times as the number of versions of the deployed application. A deployment version refers to the output of a particular transformation, using specific configurations (e.g. with certain capabilities enabled or disabled), for purposes such as testing or fine tuning. In contrast, in distributed systems, class-load-time transformation, which affects the performance of the running application, needs to be (re-)executed in each participating node. Despite the possibility of sharing/distributing transformed classes among nodes to avoid repeating the same transformation, such a solution is not always applicable to system classes due to potential incompatibility between the different versions of JVMs running on different nodes. Runtime performance impact is a major concern in this thesis, since as shown in Appendix E, the supported adaptation capabilities require complex and substantial transformations (e.g. generating classes for proxies as described in section 6.2), which could potentially cripple the performance of an application. This is especially relevant in pervasive environments given the likelihood that the tar-

get machine (the machine to which the application gets deployed) is constrained in terms of hardware resources, e.g. CPU or memory.

Furthermore, online (i.e. class-load-time) transformation of system classes (i.e. classes belonging to the standard Java API) is only possible in JVM implementations/versions providing low-level instrumentation support, thereby reducing the portability of such an approach. Additionally, such a transformation approach does not support early syntax/code verification, which is important in distributed/mobile applications and applications in similar domains such as J2EE/EJB [172], because the deployment of such applications is often a time consuming process, in which case it is important for the produced code to be syntactically verified (e.g. using a compiler) at the earliest stage possible.

Post-deploy-time transformation shares similar limitations to the class-loading approach, which include the re-execution of transformations and lack of early syntax verification. Nevertheless it does not impact on runtime application performance, since it is executed prior to application execution (i.e. offline). Consequently, pre-compile-time transformation, which is performed at source code level in contrast to the other two alternatives (i.e. class-load-time and post-deploy-time), is adopted in this work for the main transformation tasks such as generating proxy-related classes, modifying implementation classes, etc.

However, the pre-compilation of system classes (i.e. classes belonging to the Java SDK/API) might present a mismatch between the compilation environment and the execution environment of the transformed application, as will be elaborated in subsequent paragraphs. Such a limitation implies that the pre-compilation approach is not always appropriate for transformations concerning parent classes (of a proxy class or an implementation class) since as mentioned in section 6.2.1, parent classes might belong to external libraries (including the Java API) rather than the application itself.

For subsequent discussions, the term *compilation environment* is used to refer to a Java compiler and the associated Java API version, whereas *execution environment* refers to a JVM implementation and the Java API version used for application execution.

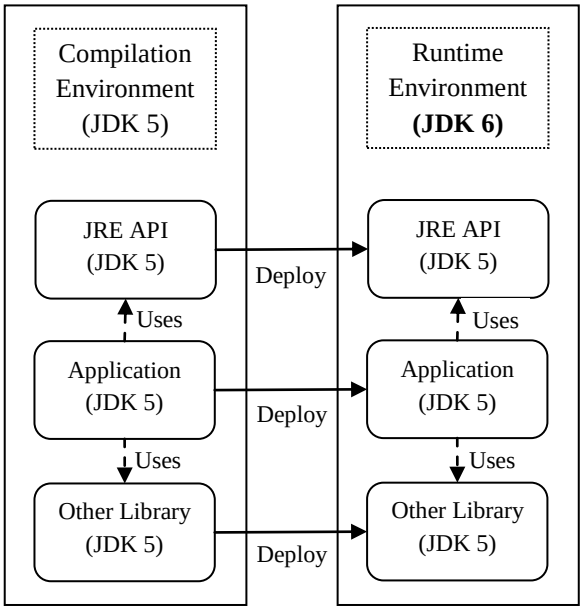

Figure 7-1. Normal Deployment Scenario

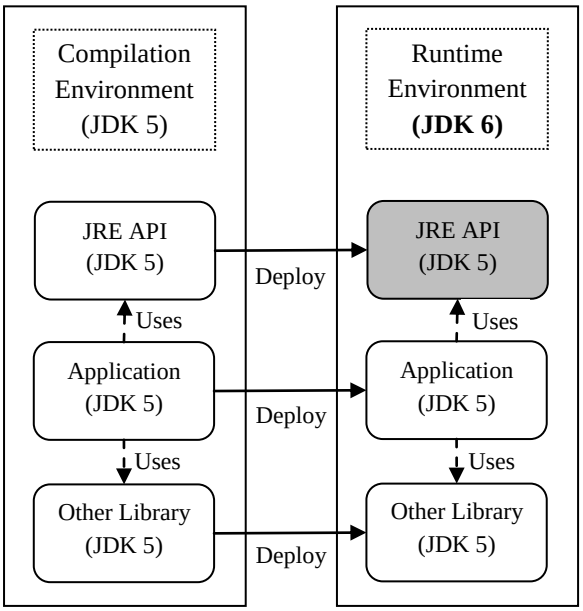

Figure 7-2. Transformation Scenario

The aforementioned limitation of the pre-compile approach, which often occurs in heterogeneous computing environments, is illustrated in an example deployment scenario shown in Figure 7-1 and Figure 7-2, where application code is compiled in a Java 5 environment and then deployed/executed in a Java 6 environment. Note that it is difficult to prevent such environments since application partitioning requires collaboration between multiple machines which are likely heterogeneous. Such a configuration does not present any problem in a typical scenario shown in Figure 7-1 (i.e. without transformation), since newer Java specifications are backward-compatible with older specifications. On the other hand, as shown in Figure 7-2, the transformation of system classes (which is applicable to some of the direct/indirect parent classes of a proxied class, as discussed in section 6.2) presents class linking errors, caused by any the following scenarios.

Firstly, deploying a subset of system classes (i.e. only those that were transformed) likely causes linking errors since the classes in the execution environment (i.e. Java 6) do not communicate in the same way (i.e. by invoking the same methods) as the classes from the compilation environment (i.e. Java 5). This problem exists even with the presence of well-defined backward-compatible Java APIs, since APIs only define classes and methods that are accessible externally, but do not define those intended to be used internally between system classes. Furthermore, deploying the entire set of system classes is not a viable option, because some system classes (e.g. AWT classes) interact closely with the JVM, thereby causing incompatibility between the transformed classes and the JVM. Additionally, such a problem occurs not only when the versions of the compilation and execution environments differ (e.g. Java 5 versus Java 6), but also when implementations differ (e.g. Sun JDK versus IBM J9).

Consequently, as a rule of thumb, the transformation of system classes should be performed on each target machine as opposed to a single source machine, thus providing the motivation for adopting the post-deployment approach (to complement the pre-compilation approach), which as previously mentioned, is preferred over the class-loading approach due to runtime efficiency. The complete deployment life cycle of an adaptive application using the code transformation techniques described in this thesis, as facilitated by both pre-compile-time and post-deploy-time transformation is discussed in Appendix D. Note that the majority of code transformation described in this thesis is performed at pre-compile time, and is thus the focus of subsequent sections.

7.1.4 Transformation Techniques

This section considers a variety of techniques and tools for transforming source code, especially that written in Java. One option for performing simple transformations is using lexical techniques/tools, such as grep, awk, etc., however such an approach is limited in its ability to express complex logic and thus is unsuitable for complex transformations such as those addressed in section 6.2.

An alternative is to use direct transformation whereby output code is generated while the original code is being parsed, e.g. by a grammar-based parser generated using tools such as Javacc [89] or Antlr [7]. However, such an approach may not be desirable due to the development complexity resulting from mixing parsing logic and transformation/generation code. Another disadvantage is the overhead of repeatedly parsing the same input code for the generation of multiple code artefacts, which although is applicable to this work since the adopted proxy solution requires multiple classes (e.g. proxy class, domain class) to be generated from a single (proxied) class (as discussed in section 6.2), is not a runtime-performance concern since source code transformation is performed at pre-compile time.

Another alternative is to use a specialised program transformation system, such as TXL [43], Stratego/XT [183] [27], and DMS [18], which although offering rich support for code analysis, requires significant understanding of the adopted language syntaxes and paradigms. On the other hand, Extensible Stylesheet Language Transformation (XSLT), an XML-based transformation system, is favoured in this thesis due to XML being an open and widely adopted standard, thus providing advantages in terms of the availability of supporting tools; technologies; documentation; and communities.

Note that although XSLT is integral to the presented transformation solution (as demonstrated in the pre-compilation process presented in Appendix B), the solution should be applicable to other transformation engines, especially those adopting a similar approach, i.e. the *intermediate-representation approach*. In this approach, Java source code is first translated

into an intermediate representation, commonly known as a *parse tree* (e.g. in XML format). Next, the parse tree is analysed, processed, and transformed into a new parse tree, which is then converted back into Java code, producing a modified application (e.g. containing new capabilities). This process can be modularised in order to allow some of the involved sub-tasks (e.g. injection of certain capabilities) to be skipped, repeated, etc. depending on deployment requirements or preferences, as discussed in Appendix B. In comparison to Aspect Oriented Programming (AOP) [97] technologies such as AspectJ [10], which aim to provide convenience to developers by hiding the detail of the transformation, the parse-tree transformation approach provides full control over the transformation process. Such control is necessary to facilitate the various types of transformation described in this thesis, which as demonstrated in Appendix E, include generating new code artefacts (i.e. classes, interfaces), changing class structures (e.g. inheritance, class name, methods), modifying class content (e.g. statements), etc.

Note that in practice, parse trees for Java source code can be represented in a number of formats, including tool-specific Abstract Syntax Trees (AST) exposed for external/public use, such as Antlr AST [8] and Eclipse AST [103], with each format offering potential benefits. For example, Antlr provides a simple mechanism for traversing and transforming the parse tree using compact syntax, whereas eclipse AST provides powerful element bindings allowing easy extraction of type information (e.g. class type) for each element in the code (e.g. variable). Nevertheless, JavaML [12], which is an XML-based representation, was used instead due to the previously mentioned benefits of XML and due the sufficiently rich and detailed information that can be expressed. The JavaML specification fully complies with Java 1.4 and since it is open source, can easily be extended to support later versions of Java. Open source libraries and XSLT style sheets for converting Java code into JavaML and vice versa are also available (e.g. Java2XML project [88]).

Although XML-based representations such as JavaML are not the most efficient, JavaML was chosen since the previously mentioned benefits were considered more important than efficiency, especially because the transformation is done statically at compile-time (i.e. does not affect the running application). In the case where compile-time efficiency was a primary concern, a binary intermediate representation could be used instead [6] at the expense of human-readability.

7.1.5 Generation of Code Fragments

The transformation of a class involves locating specific parts (i.e. code fragments) of the class for the purpose of injecting new code or modifying existing code. In either case, new code

[8] Antlr is a parser generator while Antlr AST is the tree that is (optionally) generated by the parser.

needs to be generated before it can be used for injection or as a replacement for old code. Such a task is generally difficult to achieve in the specific language used by the adopted transformation system, especially in the case where the amount of generated code is significant, a limitation that also applies to the adopted approach due to the verbosity of XSLT style sheets, which are expressed in XML.

Consequently, existing transformation systems, such as TXL and Stratego/XT, support *concrete syntax* [182], which allows the syntax of the subject language (e.g. Java) to be used as opposed to the specific syntax used by the transformation system, which is known as *abstract syntax*. Similarly, Antlr uses a templating engine, StringTemplate [166], which allows arbitrary code to be generated in plain text. However, such a feature/approach is not directly applicable to transformation that works on an intermediate representation (e.g. Eclipse AST or JavaML), since the generated textual code is not compatible with the representation. Consequently, before generated code can be used (i.e. inserted into the parse tree), it needs to be first converted into a compatible format. Furthermore, since code generation often requires information collated from various sources (e.g. class information, configuration), which introduces additional complexity, an architectural component (i.e. the Code Generator) designed to address such issues is presented in section 7.2 and discussed further in Appendix B. The Code Generator uses StringTemplate as the code generation engine due to its simplicity compared to alternatives, such as Velocity [71], Jumbo [95], etc.

7.2 Pre-compilation Architecture

This section provides an overview of the primary software components involved in the pre-compilation (i.e. transformation at *source-code* level) of an application for the purpose of injecting adaptation capabilities into the application. These components, which were designed according to the constraints, requirements, and decisions mentioned in section 7.1, play an important role in facilitating the execution of the transformation tasks required by the proxy solution described in Chapter 6, which include generating additional classes (e.g. domain classes, proxy classes); and modifying implementation and client classes.

For brevity, this section describes the newly developed transformation architecture and processes at high level, with more details in Appendix B. The discussion also serves to provide background information for the case study presented in section 7.3, which demonstrates the practical utility of the presented components, such as performing specific transformation on a class according to the specified type (e.g. mobile object class), as described later in this section. Note that *byte-code* transformation, which is minimal since it is required only for transformation concerning parent classes (due to the possibility of belonging to system

classes as mentioned in section 6.2.1), is addressed in Appendix C, which concerns the implementation of the code transformation solution in the MobJeX framework [147].

The architecture-level relationships between the various components of the transformation solution are illustrated in Figure 7-3 and summarised below, whereas the explicit interaction between these components during the transformation of application source code (i.e. the pre-compilation process), is presented in Appendix B.

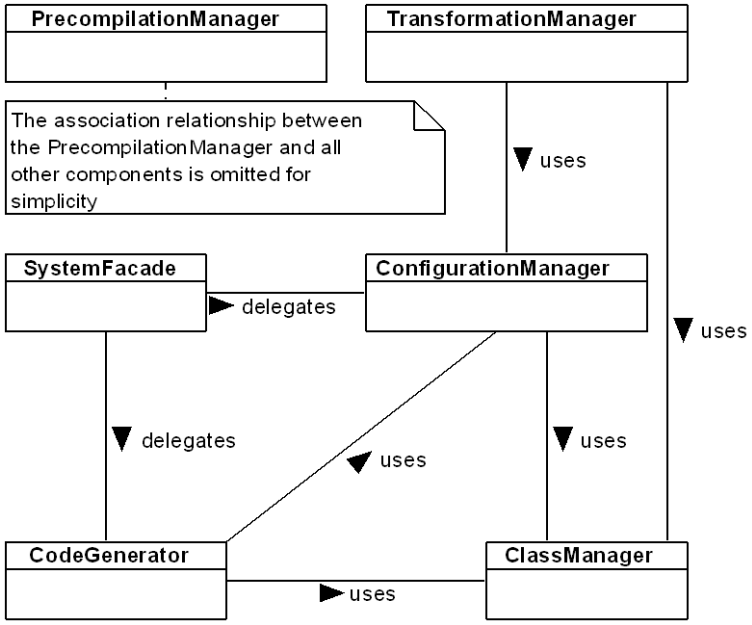

Figure 7-3. Main Pre-compilation Components

The main role of the Pre-compilation Manager is to perform high-level management tasks involved in the pre-compilation of a specific application, which include 1) managing the life cycle (e.g. initialisation) of other components (e.g. Configuration Manager), 2) performing preliminary setup operations (e.g. identifying classes requiring transformation), and 3) administering the pre-compilation of multiple deployment versions, which as discussed in section 7.1.3, are produced from the same application/code but based on different configuration (e.g. different sets of injected capabilities).

The Configuration Manager supplies deployer-specified information to other components (e.g. Transformation Manager) in order to enable customisation of capability injection (e.g. enabling/disabling metrics management capability). It supports various configuration techniques which are ordered according to their priorities. For instance, Java annotations, which are embedded in application source code for configuring specific code elements (e.g. whether a method should collect metrics upon invocation), override higher-level (thus having lower priority) configuration such as that specified in a system-wide configuration file.

The Transformation Manager is responsible for transforming individual Java classes based on the specific class type (e.g. *mobile*, *stationary*, *creator*, *normal*) assigned by the application deployer. The *mobile* type is assigned to classes whose instances can migrate be-

tween hosts, thereby involving the injection of proxies along with the included capabilities for supporting remoteness (e.g. remote invocation, data marshalling/unmarshalling, etc.), mobility (e.g. object migration, location tracking, etc.), adaptation (e.g. metrics management), etc. Since stationary objects are accessible remotely, the injection tasks applied to *stationary* classes are largely the same as mobile classes, with the primary difference being stationary classes do not require capabilities concerning mobility and adaptation. The *client* type is assigned to classes whose instances potentially instantiate or access the fields of, proxied objects (e.g. mobile or stationary objects). Finally, *normal* classes refer to classes, which do not require type-specific transformation in the way that mobile, stationary, and client classes do, but may still involve generic transformation such as renaming the package of a class (thus essentially moving the class to a different package), which is applicable across all class types.

The main role of the Code Generator is to produce code according to pre-defined templates and return the code in a format that is compatible with the adopted intermediate representation (e.g. XML/XSLT). The code templates are defined in a format supported by the adopted templating engine (e.g. StringTemplate), which allows the majority of the generated code to be pre-specified, leaving dynamic information (e.g. class names, method names) to be inserted into designated place holders during code generation. The dynamic information is acquired from various sources, which include the transformation engine (e.g. XSLT), the Configuration Manager, the Class Manager, etc.

The System Facade serves as an intermediary component which allows access from the transformation engine (e.g. XSLT) to other components in the system (e.g. the Configuration Manager), during the transformation of a class. Not only does this extra layer hide the other components in the system, but it also provides a strict and well-defined interface for accessing them, thus minimising the chance of breaking the interaction between the transformation engine (e.g. XSLT) and the system components (e.g. Java objects) upon modification (e.g. refactoring), which is important because such interaction is usually not verifiable at compile time.

The role of the Class Manager is to obtain class structural information (e.g. fields, methods, or implemented interfaces) as required by other components (e.g. the Code Generator). One main issue addressed by the Class Manager is the difference in which structural information can be obtained from the class being transformed (which belongs to the pre-compiled application) and from other classes that are referenced from the class under transformation (which may belong to an external library). In particular, although obtaining the former is straight forward as it can be easily extracted from the parse tree under transformation, acquiring the latter presents complexity in that the relevant source code might not be readily available, as addressed in Appendix B.

7.3 Case Study

This section presents a case study to demonstrate the practical applicability of the code transformation solution described in this chapter as well as the proxy solution presented in Chapter 6. A case study was chosen in favour of a formal experimental evaluation due to the difficulty of accurately measuring the concerned quality attributes (e.g. transparency, flexibility, and configurability), caused by the absence of a reliable benchmark (e.g. an existing solution for injecting the same sets of capabilities). Furthermore, an objective evaluation of development transparency is hard to achieve, since such an evaluation depends heavily on the specific characteristics of the transformed application. For example, some applications may be written using native code which could affect transparency (as discussed in section 6.2.4) or contain functionality which conflicts with the injected capabilities, whereas others might favour capability injection insofar as capabilities can be injected without breaking existing code.

Section 7.3.1 provides an overview as well as outlines the objectives, of the case study, whereas section 7.3.2 discusses the materials and procedures applied in the case study. Finally, section 7.3.3 analyses the outcome of the case study with regard to the objectives specified in section 7.3.1.

7.3.1 Overview

In order to facilitate the case study, the adopted code transformation solution was implemented in the MobJeX framework [147] (as well as a complementary transformation tool, Mobjexc), as discussed in detail in Appendix C. As a result, MobJeX supports the injection of various capabilities including those related to adaptation via application partitioning and via implementation swapping, which refers to the dynamic substitution of an implementation (e.g. in the form of objects) with another compatible implementation for the purpose of changing the functional behaviour of the application.

In addition to the many synthetic applications that have been developed for testing purposes (e.g. as used in the experiments presented in Chapter 5), MobJeX has been used to deploy and run various prototypical applications with domain-specific functionality. These include a mobile-device compatible (e.g. PDA) application for managing photographic statistics, a Taxi Dispatching System (TDS), and a prototype of a virtual world application. Since none of these applications require the unimplemented proxy transformation tasks mentioned in section 6.4, the deployment of these applications does not require manual modification of proxy-related code. Instead, only a small amount of declarative configuration was required (between 10 – 40 lines per application), although more configuration/fine-tuning could potentially result in more optimal execution.

This case study focuses on the deployment of the aforementioned virtual world application, in which the application was injected with various capabilities (e.g. metrics management) concerning adaptive application partitioning via object mobility. The primary *objectives* are to demonstrate that the newly developed capability injection solution can: 1) handle non-trivial transformations (such as those required by the proxy solution presented in Chapter 6); 2) inject non-trivial capabilities (such as those required by adaptive application partitioning); and 3) achieve the objectives stated in items 1 and 2 with minimal human intervention (e.g. without requiring manual modification of application code).

7.3.2 Materials and Procedure

A prototype of a virtual world application, which was written without any networking capabilities, was used as the base application for the injection of capabilities supported by MobJeX via code transformation. The main goal of the transformation was to provide two main application domain level functionalities: 1) network-based multiplayer functionality, which was achieved by employing object mobility to distribute user interface objects to the machines used by the participating players and 2) dynamic migration of objects based on the execution context, e.g. resource utilisation of the machines, geographical location of the players, etc., in order to improve application response time. The latter can be achieved manually via an interactive administration interface or automatically using an adaptation solution such as that presented in Chapter 3

The transformation goal was achieved via the injection of proxies containing the required capabilities, which included object mobility, metrics management, concurrency management, error handling, and object clustering. On the other hand, decision making algorithms (which include the original and the proposed algorithms) were implemented entirely in the MobJeX middleware, thereby requiring no additional code injection. *Object mobility* consists of several sub-capabilities, such as location tracking and remote communication, which are concerned with providing client/caller objects (through proxies) the ability to locate and communicate with a mobile object.

Metrics management, as addressed in Chapter 4, enables the collection and management of both application-specific (e.g. execution time, number of invocations) and environmental information (e.g. processor usage, network usage). The former, which reflects the behaviour of the application and the user/player, is the main concern of the injection, since the latter, which reflects external (execution) conditions, can be supported entirely by the MobJeX middleware independently of application awareness.

The injection also concerns functionality for managing *synchronisation* between concurrent application and middleware operations, e.g. migrating an object while its state is chang-

ing. An *error handling* mechanism is especially relevant since there is a greater likelihood of failure in remote communication. Furthermore, *object clustering* is beneficial for objects that are tightly coupled (e.g. AWT/Swing Frames and Panels) and hence are always located in the same host. A more detailed discussion on the advantages of object clustering is provided in Appendix A.

Despite the ideal case wherein an application could be developed independently of the injected capabilities, to a certain extent, the virtual world application was designed with object mobility in mind. One example is that the application was designed around the Model-View-Controller (MVC) pattern, which enabled modular separation of models and views, thereby allowing objects of the same type to be clustered and loosely-coupled objects to be migrated to and thus executed in, different hosts. Another example is that although the application did not support the remote capability, it *did* provide basic multiplayer functionality, such as chatting or character interaction, on a single machine via multithreading. Such a design would have little practical utility in the absence of the injected distribution capability.

The code transformation was configured to produce two sets of classes: one with metrics management and one without metrics management, with the intention that when automatic adaptation (by the adaptation engine) was needed, the version with metrics management would be used. On the other hand, when manual object migration (by a system administrator) was more suitable, the other version, which is more efficient since there is no metrics management overhead, would be used. Therefore, one of these two sets of classes was placed in a different java package in order to avoid naming conflicts thus utilising the automatic package renaming transformation mentioned in section 7.2.

Although the virtual world application was designed as a demonstrator and test case rather than a fully functioning application, it was still architecturally representative of a real world system, even if not of commercial scale and functionality. The original virtual world application consisted of 35 code artefacts (i.e. classes and interfaces) containing a total of 1853 lines of code (excluding empty lines and comments).

Nine classes were configured to be *mobile object* classes, while four classes were tagged as *creators* (introduced in section 7.2). The rest were tagged as *normal* (by default). Out of the nine *mobile object* classes, five classes were assigned into the same cluster in order to ensure that these classes always migrate together, whereas the rest of the classes were able to migrate independently. In addition, in the metrics-enabled injection, four classes were configured (using Java annotations) to disable metrics collection/management for reasons of efficiency. This is an example of the application of domain-specific knowledge for optimising injected capabilities (in this case to reduce metrics management overhead, e.g. memory usage), in which the collection of metrics for certain classes is perceived as unimportant be-

cause it was known in advance that the instances of these classes are infrequently invoked, thereby not having as much impact on application performance as other classes.

The results of the transformation were first verified manually by human inspection at the source code level to ensure that code was produced according to the configuration, e.g. whether the required artefacts such as proxies were generated correctly; whether metrics management code should exist; etc. Next, the produced code was deployed to three machines and post-deployment transformation (which as mentioned in section 7.1.3 is performed at byte-code level) was executed on each target machine (using the automatically generated deployment scripts discussed in Appendix D). The resulting code was then launched to ensure that the application ran and behaved as expected with the injected capabilities, which provided automatic distribution of multiple players over a network, transparently creating a distributed, real-time, multi-user application.

7.3.3 Analysis

The deployment arrangement described in section 7.3.2 required only 40 lines of configuration information since most of the default configuration values were already suitable for the code transformation. The transformation produced 142 code artefacts, half of which belong to the version with metrics management, while the other half belongs to the version without metrics management. The former consists of 10912 lines of code, while the latter consists of 10178 lines of code.

In the transformed application, a new player window could be created and migrated to a specific machine upon a request made by a newly participating user/player. In this case study, a player window was requested from each of the three target machines (mentioned in section 7.3.2) to simulate the participation of three players. Once the requesting machine received the window, the player could control its character in the virtual world and interact with other objects in the world.

Even though adaptive object mobility was also injected, its behaviour was not tested in this scenario since this required more realistic scenarios in which the application contained more computationally expensive operations (thus able to benefit from migration to less loaded machines) and had players interacting with the application simultaneously from various machines at different geographical locations (thus allowing improvement in terms of reducing communication overhead by migrating relevant objects based on player interaction in virtual space).

Nevertheless, the deployment (including the injection of capabilities) of the virtual world application, was considered a success, because the objectives mentioned in section 7.3.1 were met, as explained below:

1. The injection involved non-trivial transformations on a number of classes which covered the different proxy-inheritance cases addressed in section 6.2 (e.g. a proxied class extending another proxied class).
2. The injection involves complex capabilities as was shown by the (previously mentioned) amount of the produced code (an example of which is provided in Appendix E). Note that the generation of repetitive code has been minimised as it is one of the design decisions as discussed in section 7.1.1. In fact the majority of the injected code only served to connect to (i.e. to access the functionality provided by) the MobJeX middleware, as shown in Appendix E.
3. The deployment was almost completely automated since no additional Java code was required and thus only involved human intervention for performing configuration, setting up runtime environments, and initiating executions (e.g. post-deploy transformation, launching the application).
4. Both versions of the application were successfully deployed and launched, and were semantically/operationally correct.

Chapter 8. Summary and Conclusion

This chapter concludes this thesis by providing a summary of the research goals as well as how these goals were realised, as presented in section 8.1. Furthermore, the implication of this realisation in terms of practical benefits is discussed in section 8.2. Finally, section 8.3 discusses the main limitations of this work, which open the path for future investigation to improve certain aspects of the proposed solutions.

8.1 Summary

The goals of this research were: 1) to improve the effectiveness (in terms of performance improvement) of local-adaptation via application partitioning, 2) to automatically collect and manage metrics required by the adaptation, and 3) to facilitate the transparent development of applications supporting such adaptation. The realisation of these goals resulted in various inter-related solutions for adaptation decision making, metrics management, and capability injection.

The proposed decision making solution, which was derived from an existing solution [144] (also referred to as the *original solution*), improved the quality of decision making through the use of alternative metrics (e.g. processor usage time instead of execution time) which more accurately represent certain execution conditions (e.g. the CPU usage of an object). Decision making quality was further improved by employing additional information (e.g. to capture the interaction between individual objects), which enabled more accurate estimation of application performance. The proposed solution, which is more effective than the original adaptation solution, can improve application performance (compared to the non-adaptive execution of the same application) in scenarios involving heterogeneous and/or dynamic execution conditions (i.e. application behaviour and execution environments), as demonstrated in section 5.3.

The proposed metrics management solution was established through the identification of the different characteristics of metrics that are required by the adopted adaptation algorithm. The solution concerns metrics collection in terms of how, when, and where the relevant measurement (for obtaining certain metrics) should be performed. Other concerns relate to the delivery of collected metrics to distributed adaptation engines and the efficient representation of metrics for the purpose of capturing their temporal characteristics. The solution, which can be extended to support different adaptation solutions, was used to facilitate adaptation as demonstrated in the experiments presented in Chapter 5, which evaluated the effectiveness of the proposed solution in comparison to the original decision making algorithm.

The proposed capability injection solution was presented in two parts. The first part concerned the transparency of object proxies, the main role of which is to support automatic injection of additional functionality (e.g. metrics collection) into existing applications. Although the resulting (partial) solution does not attempt to achieve full transparency (i.e. allowing capabilities to be injected without breaking the original application in all possible scenarios), it offers major improvement in transparency compared to existing solutions as discussed in Chapter 6. Furthermore, this solution is not specific to adaptation but is applicable to any problem/situation requiring object proxies, which as reviewed in section 2.6, represents a large variety of applications.

The second part of the injection solution addressed practical issues concerning capability injection, with particular focus on facilitating the code transformation tasks required by the proposed proxy transparency solution. Additionally, the solution addressed issues which are typical to adaptive applications, such as manual customisation (e.g. fine-tuning) of injected functionality and deployment on heterogeneous platforms. Together with the proxy transparency solution, this forms a collective solution which significantly reduces the effort of developing applications supporting adaptive application partitioning, while providing flexible support for the human involvement that is still required, e.g. for configuration, fine-tuning. Although scientific evaluation of the code transformation solution was not performed (due to the difficulty of objectively measuring transparency as discussed in section 7.3), the solution has been used to automatically inject capabilities (including those related to application partitioning as well as other types of adaptation, e.g. functional adaptation via implementation/object swapping) into various applications, as discussed in section 7.3.

8.2 Practical Applicability

The proposed adaptation solution was shown (in the evaluation presented in section 5.3) to be beneficial for applications which can be decomposed into relatively detached objects or groups of objects (i.e. low interaction intensity). Consequently, the solution offers obvious benefits in the domain of component-based (e.g. Enterprise Java Beans) or service-oriented systems (e.g. those based on web services), since such systems are developed in terms of functionality which is implemented as isolated components or services. Note that even though the functionality of service-oriented systems is exposed as services, which are not necessarily compatible with the object-oriented paradigm, the internal implementation of such systems can be object-oriented.

Nevertheless, further investigation is required to fully exploit the benefits promised by adaptive application partitioning in component-oriented or system-oriented systems. For example, systems such as those using EJB, require support from specialised middleware, and

thus issues related to how adaptation functionality can be seamlessly incorporated into the middleware, need to be investigated. Furthermore, due to the common application of component-oriented or service-oriented systems in the web domain, web-specific properties such as multi-threading (i.e. client requests are served in separate threads), statelessness (i.e. client state is maintained in external storage such as a database, instead of in the application itself), and dynamic usage patterns (i.e. access frequency varies on specific times, days, etc.), should be considered. While the issue of dynamic usage patterns, which affects application behaviour, has been addressed by the ability to adapt to dynamic application behaviour (as demonstrated in the presented evaluation), multi-threading and statelessness can be exploited by applying features such as parallelism analysis (e.g. [87], [52]) and object replication (e.g. [165]), as discussed further in section 8.3.

Due to its ability in coping with heterogeneous and dynamic execution environments, the proposed adaptation solution is suitable for applications running on platforms exhibiting such characteristics, such as those concerning grid and mobile computing. The solution is likely to be particularly beneficial to applications targeted for mobile devices, since available resources are generally limited (e.g. low computational power) and volatile (e.g. unstable wireless signal strength). A general approach to address resource limitation, is to have the majority of the objects of a resource-demanding application (e.g. video manipulation software) execute on more powerful machines, but leaving the graphical interface on the mobile device for user interaction. Adaptation allows objects to be relocated later as a result of changes in machine conditions (e.g. machine load) as well as user location (e.g. longer distances cause slower network access).

Despite the expected suitability in specific domains, the proposed solution concerning adaptation and metrics management, should be beneficial to many applications, because the manner in which metrics are collected and used in decision making, is generic and thus is potentially applicable to any object-oriented application. Furthermore, the accompanying solution for facilitating application development through capability injection, is generic such that not only is it potentially applicable to any applications written in Java, it can also be used to inject capabilities required by future extension (e.g. new functionality, improved adaptation algorithms), be it concerning object mobility or other types of object-based adaptation (e.g. dynamic object swapping, replication of objects).

Since the proposed solutions concerning adaptation, metrics management, and capability injection as well as the complementary functionality (e.g. object mobility, concurrency management) have been implemented and tested in the MobJeX framework, the solutions are readily usable through the development and runtime support (i.e. pre-compiler and middleware components) provided by the framework. Consequently, not only does the framework facilitate the collective injection of related capabilities (for example adaptive application par-

titioning as well as the supporting capabilities, e.g. metrics management), but it also supports the injection of capabilities independently of each other. For example, object mobility can be injected without the presence of adaptation in order to enable networking capability (for an originally non-distributed application) or to allow explicit user-driven (i.e. non-automated) object migration. Such support from the framework can facilitate future studies on related topics (e.g. adaptation, metrics, and mobility) in terms of significantly reducing the involved implementation work.

8.3 Limitations and Future Work

Despite the promising results from the proposed local-adaptation algorithm, a number of extensions can be applied in order to improve its effectiveness in different execution scenarios, such as those involving multi-threaded applications and/or multi-core CPUs. In particular, as the adoption of multi-threading increases (even for the development of regular non-parallel applications, e.g. desktop applications) due to the emergence of multi-core systems, there is increasing appeal to incorporate parallelism analysis in decision making, similar to the approach proposed in [87]. On a related note, decision making can also be extended to better exploit multi-core machines by taking into account specific characteristics of multi-core systems (e.g. the probability in which a particular thread execution is migrated to a different core) and individually analysing its state (e.g. the degree of load).

Another possible improvement is the consideration of additional hardware resources, such as disk storage, the investigation of which was left for future work due to the complexity involved wherein different types of disk operations (e.g. read/write, cached/direct read) have a different impact on application performance. Additional adaptation goals, such as load balancing (as proposed in [144]), improving application robustness (using migration and replication), or conserving energy (through analysis of battery state, consumption rate, etc.), can also be introduced in order to enable multi-goal decision making, which is supported by the adopted solution through the use of multiple indicators as explained in section 3.1.

The aforementioned improvements primarily involve extending the proposed algorithm, which consequently requires extension to the metrics management solution to support additional metrics such as those related to disk I/O or energy consumption. On the other hand, as will be explained in subsequent paragraphs, improvements such as those for enabling early optimisation of object placement as well as for enabling adaptation of collective objects, potentially require significant changes to the calculation of adaptation scores.

Early optimisation of object placement is concerned with bringing the application to a more optimal layout as early as possible, instead of relying on the traditionally adopted *continuous adaptation* (introduced in section 2.1), which only takes place once sufficient metrics

have been collected. One envisaged approach is similar to that adopted in [186], in which object migration is performed at application start-up, based on software metrics obtained via code analysis (prior to application execution). Host resource metrics can also be collected prior to application execution by profiling the capacity and average load of participating machines (independently of the application).

Adaptation of collective objects addresses a limitation of the proposed algorithm in which objects are not migrated due to their insignificant contribution to application performance when analysed individually, despite their potential significance when migrated as a group. For example, an object which interacts heavily with a group of objects, is of low migration priority due to the implied high network usage (after it is migrated to a separate machine), which can be prevented by migrating all of these objects collectively. One particular issue requiring investigation is how objects are clustered dynamically (at runtime) in order to allow them to be migrated as cohesive groups.

Future work could also involve an investigation of alternative techniques to the adopted scoring system for making adaptation decisions, which include the adoption of inference and learning techniques such as Bayesian network [92]. Another possible investigation on the subject of adaptation decision making is the replication of local adaptation engines, which serves to alleviate the burden of an adaptation engine when all objects are migrated to the same machine.

A potential improvement to the proposed metrics management solution is the decentralisation of the metrics delivery process, allowing metrics to be exchanged between machines, without a centralised context server, which is a potential performance bottleneck and a single point of failure. However, in order to adopt such an approach, the inherent issue of network congestion which is caused by the flooding of exchanged information needs to be addressed [105]. Another metrics-related improvement concerns the prediction of future metrics, which can be realised by extending metrics representations (e.g. exponentially weighted moving average) to take into account seasonal changes as discussed in section 5.3.4. A more sophisticated approach is to apply *function approximation* through the use of techniques such as artificial neural networks [113].

In terms of capability injection, extensions can be made to support injection concerning applications developed using other object-oriented languages, such as C# (.NET), through the modification of the proposed solutions concerning proxy transparency (Chapter 6) as well as the required code transformation (Chapter 7). The former is not expected to involve significant effort due to the conceptual similarity between Java and languages such as C#, whereas the latter is particularly challenging due to the unique compilation processes applied by different technologies (e.g. .NET generates assemblies instead of class byte code), thereby potentially requiring a significantly different code transformation solution.

Appendix A. Object Clustering

Object clustering, which allows mobile objects to be grouped as a cluster, ensures that objects belonging to the same cluster, always migrate together. Such a scenario implies that proxies and the included functionality, such as remote invocation and metrics collection capabilities, are not required for communication between objects in the same cluster.

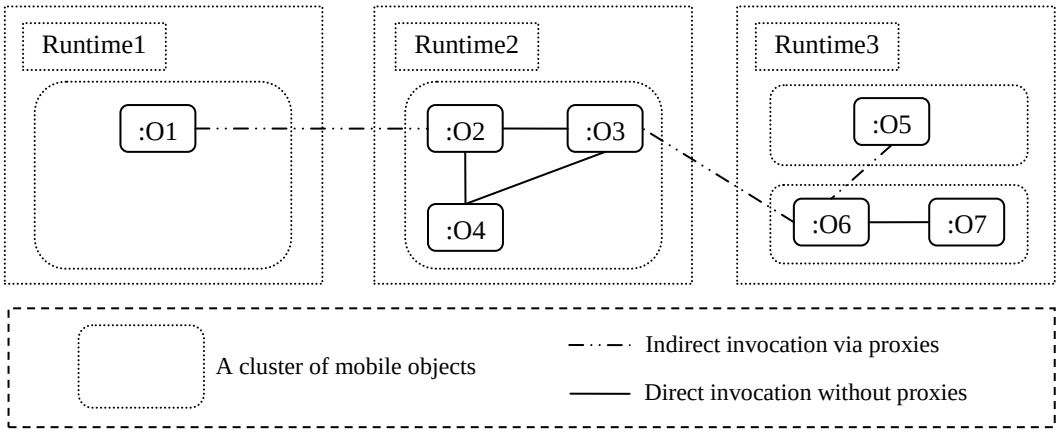

Fig. A-1. A System Consisting of Four Object Clusters

Remote invocation is not required because these objects are always local to each other (i.e. running on the same process/machine) as illustrated in Fig. A-1. Furthermore, collecting metrics related to object invocation such as IF (Invocation Frequency), PUT (Processor Usage Time), and SSP (Size of Serialised Parameters), is not necessary, because these objects will never be separated anyway, therefore IF and SSP, which are used to measure the degree of remote interaction between objects (as discussed in section 3.2), are not required. Furthermore, the measurement of PUT for individual objects within the cluster (e.g. between O2, O3, and O4) is not necessary, because adaptation concerns the overall PUT (i.e. since objects move as a group), which can be measured when O1 invokes O2 as demonstrated in Fig. A-2.

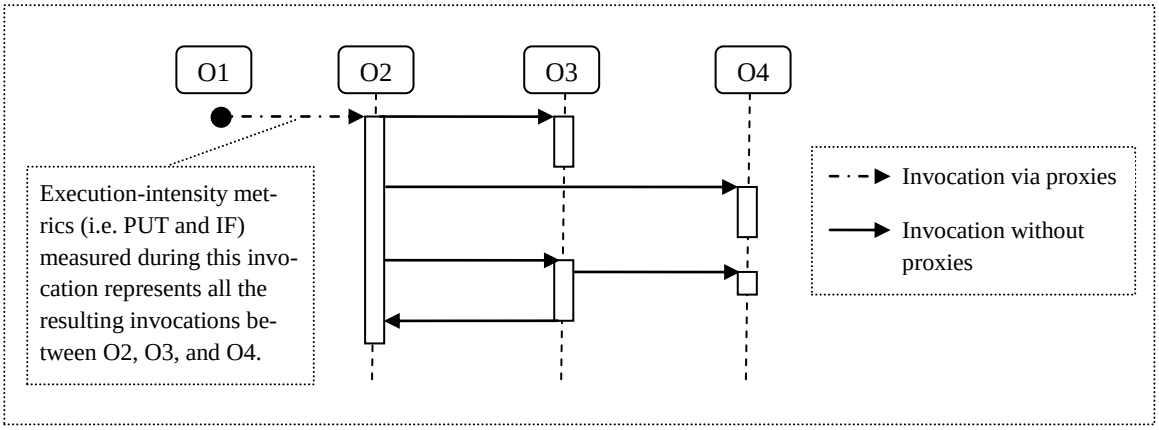

Fig. A-2. An Illustration of Metrics Collection within an Object Cluster

One useful application of object clustering is to group objects representing GUI (Graphical User Interface) components, such as those belonging to Java AWT (Abstract Windowing Toolkit) or the derivative Swing Framework. For instance, a window (e.g. `java.awt.Frame`) generally contains sub-components (e.g. `Panels` and `Buttons`), which in most cases, are integral to the main window, and therefore inseparable. Although the solution presented in section 6.2 allows proxies to be used transparently (i.e. without affecting semantics) for bridging communication between these objects (e.g. `Frames`, `Panels`, `Buttons`), removing unnecessary proxies could result in significant improvement in terms of processing and memory overheads. Such overheads could be significant since some classes in the AWT/Swing frameworks contain many methods as outlined in section 6.3. Furthermore, although the generated proxies can be configured to include only particular methods for efficiency, using object clustering removes this need thus promoting deployment simplicity.

Appendix B. Pre-compilation Process

This section presents a pre-compilation process based upon the constraints, requirements, and decisions introduced in section 7.1, which covers the majority of the transformation tasks required by the proxy solution described in Chapter 6, such as the generation of additional classes (e.g. domain classes, proxy classes), and the modification of implementation and client classes. Other transformation, which concerns parent classes, should be performed at post-deploy time and thus at byte-code level as addressed in Appendix C, since as mentioned in section 6.2.1, these classes might belong to an external library, and thus are potential system classes.

Subsequent discussion focuses on generic architectural challenges as opposed to implementation-related and MobJeX-specific issues, which are instead addressed in Appendix C. That said, the adoption of certain technologies and tools is discussed where necessary in order to facilitate the presentation of the complete process.

1. Architectural Components

Fig. B-1 illustrates the relationship between the main software *components* involved in the architecture of the presented transformation process. The roles of these components in terms of supporting application transformation (i.e. capability injection) as well as the configurability of such transformation are the main focus of subsequent discussions.

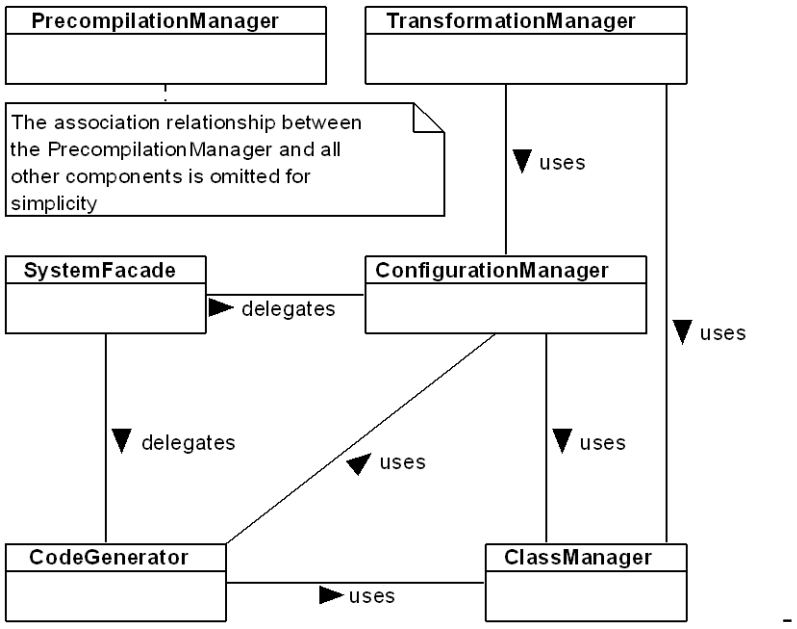

Fig. B-1. Main Pre-compilation Components

Pre-compilation Manager

The main role of the Pre-compilation Manager is to perform high-level management tasks involved in the pre-compilation of a specific application as described below. It uses parameters supplied by the application deployer (e.g. input files/directories, output directory, etc.) to perform preliminary setup operations such as verifying the provided file/directory paths, identifying classes requiring transformation, etc. The Pre-compilation Manager also administers the pre-compilation of multiple deployment versions, which as discussed in section 7.1.3, are produced from the same application/code but based on different configuration (for example in terms of which specific capabilities are injected) as supplied by the deployer.

However, instead of directly executing specific pre-compilation tasks, the Pre-compilation Manager coordinates the collaboration between different pre-compilation system components, namely the *Configuration Manager*, the *Transformation Manager*, the *Code Generator* and the *Class Manager*, the roles of which are discussed in subsequent sections. Consequently, it is responsible for managing the lifecycle of the components, in terms of their initialisation and re-initialisation (for subsequent transformation), which requires particular care since certain initialisations have to be done in specific orders due to the dependencies between components. For example, the initialisation/execution of the Transformation Manager requires prior initialisation of the Configuration Manager, because the transformation of a class requires configuration information provided by the Configuration Manager.

Configuration Manager

The Configuration Manager holds an important role in capability injection because it enables a flexible injection whereby application deployers can control the added capabilities (e.g. fine-tune their behaviour, enable/disable capabilities) with reduced effort (i.e. minimal authoring of logic/code). The adopted configuration mechanism is based on a concept popularised by Enterprise Java Beans (EJB) 3.0 [171], "Configuration by Exception", which aims to reduce deployment effort by requiring explicit configuration (using Java annotations) only when the standard behaviour (of EJB components) needs to be overridden. In this approach, Java annotations are used to provide configuration information in the form of metadata attached to specific code elements, such as classes, methods, and variables.

Nonetheless, a more powerful approach is needed to allow configuration at higher levels, such as global configuration, which unless overridden by lower level configurations, affects all pre-compilation executions (as opposed to only specific code elements). Furthermore, although source code level configuration is appropriate for the configuration of application classes (i.e. those belonging to the application), it might not be the case for external classes for which the source code might not be available, thus providing a motivation for a configuration approach that can be applied independently from source code.

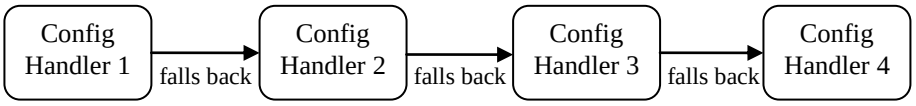

Fig. B-2. An Example of Chaining Configuration Model

Consequently, in order to support varying configuration approaches, this thesis applies a chaining model based on the well-known chain of responsibility design pattern [60]. In this model, each supported configuration level is managed by a handler in the chain, which spans from the lowest-level configuration handler to the highest, thereby prioritising the more specialised lower-level configurations. Not only does this provide an elegant functional solution, it also allows handlers to be independently inserted into the chain in order to add another level of configuration.

Obtaining a particular configuration property from the Configuration Manager begins with the execution of the lowest-level handler (e.g. handler 1 as shown in Fig. B-2). If the first handler cannot determine the property value (e.g. not specified), the next handler (e.g. handler 2) will be executed. The process continues until a value can be determined or until the end of the chain is reached, which constitutes a deployment error, since default values should be specified in the highest-level configuration (e.g. handler 4). Appendix C discusses the handlers that have been implemented and proven useful in various deployment scenarios related to the work described in this thesis.

Transformation Manager

The Transformation Manager is responsible for applying transformations to Java classes at source-code level. An important characteristic of the transformation is that it does not always produce one output for a single input. Multiple code artefacts may be produced as a result of the transformation of a given class, as demonstrated in the proxy solution proposed in section 6.2, in which not only is the original class modified, but at the minimum a separate proxy class and domain interface are generated. Furthermore, two additional artefacts (i.e. local and remote interfaces) are required for the injection of capabilities (e.g. metrics collection, object mobility) addressed in this work as discussed in Appendix E.

In the proposed solution, class transformation is decomposed into a series of modular tasks in order to promote reusability as described below. The solution is based on the *strategy* design pattern [60], whereby multiple transformation strategies were implemented for different groups of capabilities. In this approach, a class (generally considered as the smallest transformable unit), is assigned a *type*, which may be done manually via configuration or automatically via code analysis. The class *types* of relevance to this thesis, such as the *mobile object* type, will be discussed in detail in Appendix C.

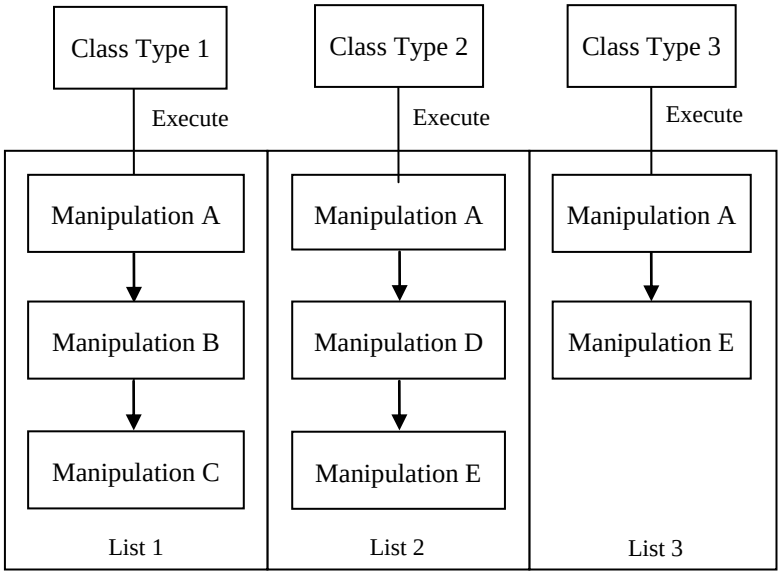

Fig. B-3. Relationships between Types/Strategies, Manipulations, and Manipulation Lists

As depicted in Fig. B-3, each class *type* corresponds to a transformation strategy which defines a *manipulation list* consisting of a series of *manipulation tasks* required for the class. For clarity, the rest of this discussion uses the term *transformation* to refer to the entire process of transforming a class based on its type, which consists of multiple individual *manipulation* tasks.

In its simplest form, each manipulation task serves to inject a specific capability into the input class although in practice this is not always the case. For instance, a task can be written to move a Java class to a different package for purposes such as avoiding naming conflicts, and therefore not involving capability injection at all. Furthermore, although the design of manipulation tasks should ideally be modular in order to promote reuse and simplify the task of enabling/disabling optional capabilities (based on configuration), some capabilities are implemented into a single manipulation task due to the close interaction between these capabilities.

An example is the injection of error handling mechanisms for addressing the reliability issues/challenges (e.g. machine disconnection) presented by other capabilities (e.g. object mobility), which only makes sense when complemented with the injection of the relevant capabilities (e.g. object mobility). Arguably, in this case, the injected error handling mechanisms can be considered as part of object mobility, as is the case with other sub-capabilities associated with object mobility, such as remote communication, object location tracking, pass-by-reference emulation in remote calls, etc.

Each manipulation task is carried out by modifying the (in-memory) XML document representing the class to be manipulated, which involves one or more XSLT style sheets depending on the number of code artefacts to be generated. Since manipulation tasks can easily be

added to the *manipulation list;* taken out from the list*;* and queued in a different order, new transformation strategies (which correspond to particular class types) can be introduced and existing strategies can be extended with minimal effort, although in practice, there may be issues arising from the interaction between capabilities (e.g. conflicting functionality, missing dependencies), as has been investigated in previous work on feature or aspect interaction [48] [47] [104] [149]. However, such issues, which are domain-specific, are not of concern in this work because they have been prevented either by applying manipulation tasks in a specific order or by implementing tightly-coupled capabilities into a single manipulation task (as mentioned in the previous paragraph).

Code Generator

The main role of the Code Generator is to produce code according to pre-defined templates and return the code in a format that is compatible with the adopted intermediate representation (e.g. XML/XSLT). The code templates are defined in a format supported by the adopted templating engine (e.g. StringTemplate), which allows the majority of the generated code to be pre-specified, leaving dynamic information (e.g. class names, method names) to be inserted into designated place holders during code generation. The dynamic information is acquired from various sources, which include the transformation engine (e.g. XSLT), the Configuration Manager, the Class Manager, etc. Since the chosen intermediate representation for transformation is XML-based (i.e. JavaML), the code produced by StringTemplate, which is in plain text, should be converted into an XML node or node list (according to the structure defined by the JavaML specification) in order to allow generated code to be inserted into the main parse tree, which represents the class that is being transformed.

System Facade

The System Facade serves as an intermediary component which allows access from the transformation engine (e.g. XSLT) to other components in the system (e.g. the Configuration Manager), during the transformation of a class. Not only does this extra layer hide the other components in the system, but it also provides a strict and well-defined interface for accessing them, thus minimising the chance of breaking the interaction between the transformation engine (e.g. XSLT) and the system components (e.g. Java objects) upon modification (e.g. refactoring), which is important because such interaction is usually not verifiable at compile time. Furthermore, since the facade decouples the details of class transformation from the rest of the system, it reduces the impact on other system components when switching to a different representation (e.g. JavaML) or technologies (e.g. XSLT).

In addition to providing necessary information (e.g. configuration properties, generated code) to the transformation engine, the System Facade may also receive notification of the

current transformation state (e.g. current class name, package name) and caches it for later use (e.g. subsequent operations and transformations). In doing so, the facade may delegate to other components for complex operations such as deciding the inclusion of functionality/code based on configuration information (with the help of the Configuration Manager), generating code for insertion into an existing method body (with the help of the Code Generator), etc.

Class Manager

The role of the Class Manager is to obtain class structural information (e.g. fields, methods, or implemented interfaces) as required by the Configuration Manager, Transformation Manager, and Code Generator. Certain tasks, such as transformation employing the proxy inheritance solution proposed in section 6.2.1, require structural information related to the class being transformed (which belongs to the pre-compiled application) as well as referenced classes (which may belong to an external library). Although obtaining the former is straight forward as it can be easily extracted from the parse tree under transformation, acquiring the latter presents complexity in that the relevant source code might not be readily available. Consequently, since the byte code of a class (be it an application or an external class) is always accessible, for uniformity, structural information of classes is predominately acquired using the standard Java reflection API, which has a further advantage of being simpler and more efficient, because it does not require traversal of source code (or the parse tree) to retrieve basic structural information.

In order to allow an application to execute on heterogeneous machines running varying Java environments, the Class Manager has the role of ensuring that classes are transformed against the API of the environment providing the lowest common functionality (which is non-trivial as discussed further in Appendix C). For example, if the target machines run either Java 5.0 or Java 6.0, the application should be pre-compiled against the Java 5.0 API in order to ensure it runs on all the machines. It should be noted that such a scenario will only work if the pre-compiler has access to the Java 5.0 API, and the original application is written for Java 5.0 or lower, thus not requiring newer functionality. Another responsibility of the Class Manager is to keep track of modified class information (e.g. new packages, renamed classes) in order to provide consistent information to other components.

2. Overall Process

This section discusses the detail of the proposed pre-compilation process by bringing together the previously mentioned components (i.e. Pre-compilation Manager, Configuration Manager, Transformation Manager, Code Generator, System Facade, and Class Manager) and describing their roles in the execution of the involved tasks. For completeness, the discussion

also includes specific implementation details, such as the tools or libraries used in certain steps/tasks (e.g. converting Java source code to JavaML files). Fig. B-4 illustrates a high-level overview of the pre-compilation process.

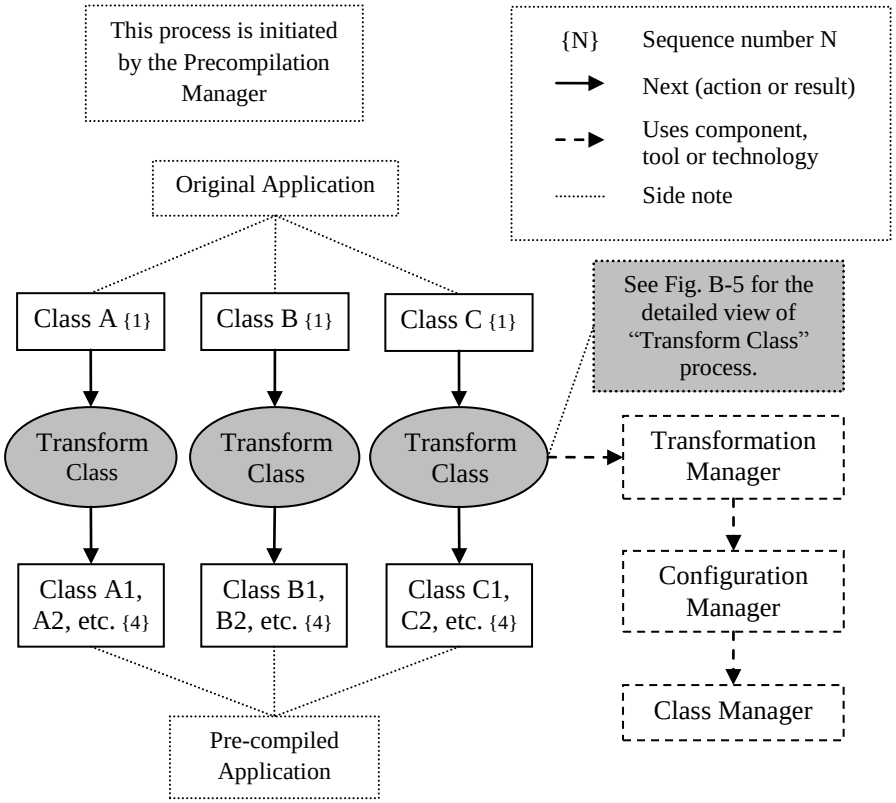

Fig. B-4. Overview of the Pre-compilation Process

The pre-compilation process is initiated with the execution of the Pre-compilation Manager, which immediately initialises the Configuration Manager and the Class Manager to load configuration information such as input files/directories, output directory, and class type (e.g. mobile object), as provided by the application deployer in the form of command line arguments, configuration files, and/or annotated source code. Next, classes requiring transformation as well as the assigned transformation types (e.g. mobile object, stationary object) are identified by traversing annotated source code in the input directory.

Depending on the specified configuration, the Pre-compilation Manager will initiate the pre-compilation to produce one or more deployment versions. Loaded information, such as class structural information, is cached to allow reuse by subsequent pre-compilations for different deployment versions. Note that the pre-compilation of different versions may be performed in parallel if speed is a concern, but this presents complexity in the synchronisation of the cache access.

The Transformation Manager is then executed to handle the transformation of each of the identified to-be-transformed classes. For each class, the source code is converted into the in-

termediate representation (e.g. JavaML) as illustrated in Fig. B-5. A sequence of transformation tasks is then executed, which as a final result, produces one or more transformed source files per class, depending on whether multiple code artefacts (i.e. classes or interfaces) are required for the injected capabilities. For instance, in the case of classes requiring proxies, such as mobile object classes, multiple files (e.g. proxy class, domain interface) are produced as discussed in section 6.2.1.

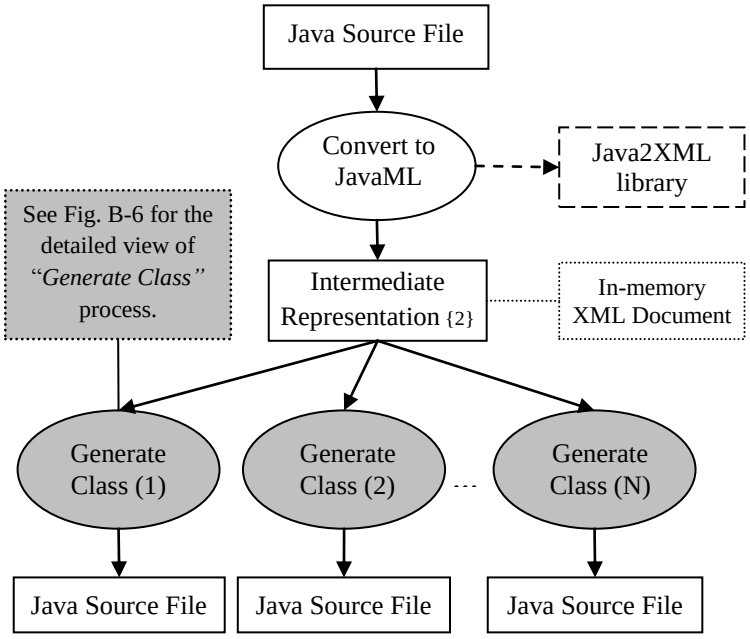

Fig. B-5. Transformation of a Java Class

As shown in Fig. B-6, a code artefact is produced by firstly performing specific transformation on the intermediate representation (e.g. JavaML) of the original code and then converting the resulting representation back to a Java source file (e.g. using the XSLT style sheet from the JavaML project[9]). Optionally, the produced code can be formatted using a Java source code beautifier to make it more readable, making it easier to modify/extend manually.

[9] JavaML project homepage: www.badros.com/greg/JavaML/

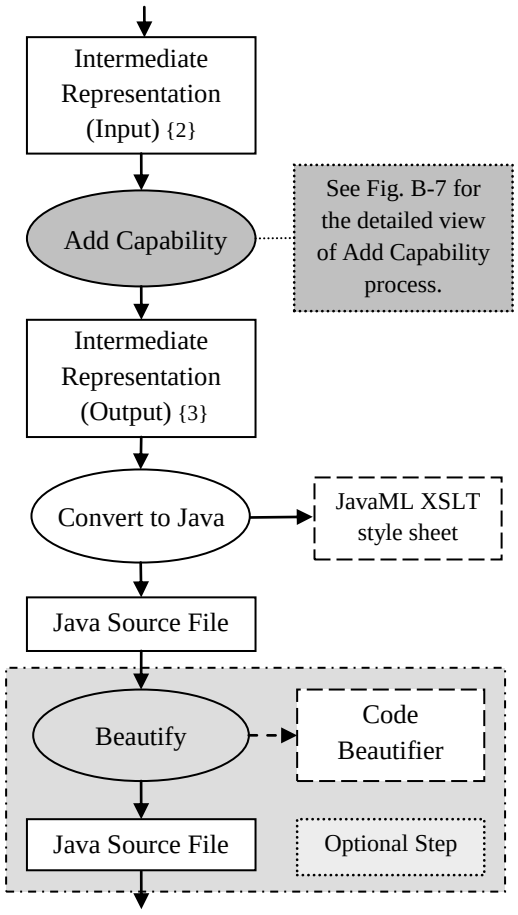

Fig. B-6. Generation of a Specific Class

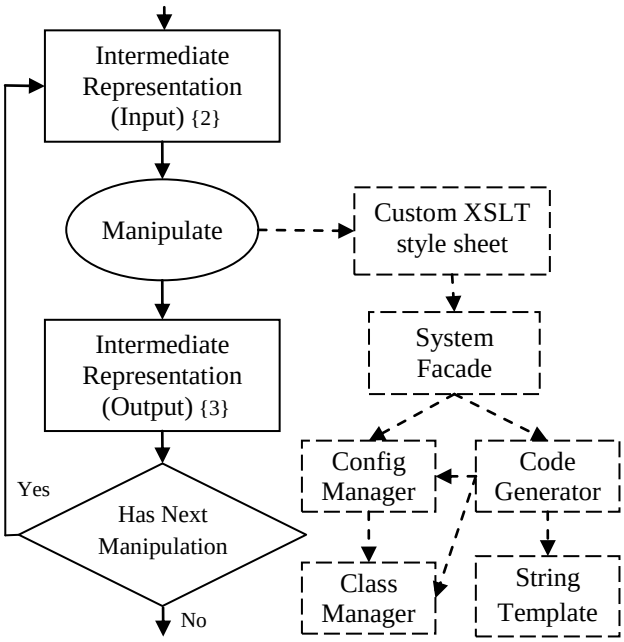

Fig. B-7. Class Manipulation Process

The detail of class transformation is shown in Fig. B-7, wherein the relevant intermediate representation is manipulated (e.g. using selected XSLT style sheets) based on the assigned class type (e.g. mobile object). During the transformation, the System Facade can be invoked as many times as required (e.g. using XSLT extension functions) to acquire configuration information (from the Configuration Manager) or to generate a substantial amount of code (with the help of the Code Generator). After each transformation, the resulting representation can be validated (e.g. against the JavaML Document Type Definition) to ensure that the produced code is syntactically correct. If further manipulation is required, such as for injecting other capabilities, the transformation step can be re-executed (e.g. using a different set of style sheets).

Appendix C. Code Transformation in MobJeX

This section discusses using specific examples of how the primary pre-compile-time transformation (i.e. pre-compilation) and the complementary post-deploy-time transformation is supported by MobJeX. Due to the extensiveness and complexity of the tasks involved in the pre-compilation process, the pre-compilation functionality is implemented in a separate tool called Mobjexc.

Subsequent discussions explain the roles of the Configuration Manager, the Transformation Manager, the Code Generator, and the Class Manager, in the specific pre-compilation supported by Mobjexc (i.e. for injecting adaptation capabilities). Furthermore, the discussions also address the implementation of the Post-deploy-time Transformer, the role of which is to manage the required additional transformation concerning parent classes, which as mentioned in Appendix B, is performed on class byte code, after application deployment.

Configuration Manager

As discussed in Appendix B, the Configuration Manager employs a chaining model consisting of multiple handlers with varying priorities. The configuration handlers that have been implemented in Mobjexc are presented in Fig. C-1, listed from the most generic (i.e. lowest priority) to the most specific (i.e. highest priority) handlers. Fig. C-1 also includes information regarding the implementation technique used for specifying the corresponding configuration, as well as the scope in which the configuration takes effect. Depending on the property that is being configured, some of the handlers are not needed. For example, the property for determining whether objects of a class should be *mobile*, is used only at class level, and is thus not applicable to methods.

Handler	Implementation Technique	Scope
Global	Global properties file	All pre-compilations
System property	JVM parameters, properties file, or `System.setProperty()`	Specific pre-compilations
Overridden class	Class annotations	Specific classes
Class	Class annotations	Specific classes
Overridden method	Method annotations	Specific methods
Method	Method annotations	Specific methods
Deployment descriptor	XML configuration file	Specific applications

Fig. C-1. Handlers in Chained Configuration Model

The main goal of having *global* configuration is to allow application deployers to specify the default/standard pre-compilation behaviour. For example, disabling *metrics management* support at this level will affect subsequent pre-compilations, thereby effectively disabling automatic adaptation of all pre-compiled applications, unless more specific/low-level configuration is specified. Nevertheless, objects of the application can still be migrated manually (i.e. through an administration interface), assuming that the support for object mobility is enabled.

Configuration using *system properties* allow configuration on a per pre-compilation basis, thus is useful where a particular pre-compilation is different from the rest. Since this configuration is more specific (thus having a higher priority) than global configuration, specified properties will override those specified in global configuration.

```
1  @Mobile
2  @DefaultProxied( { Visibility.PUBLIC, Visibility.PROTECTED })
3  public class A {
4
5    @Proxied(false)
6    public void m() {
7      ...  // Method body
8    }
9    ...  // Other declared instance methods
10 }
```

Fig. C-2. Class-level and Method-level Annotations

Class-related configuration, which includes *overridden class*, *class*, *overridden method*, and *method*, enables configuration of individual classes/methods or groups of classes/methods. *Overridden class* configuration allows a child class to inherit the configuration of the parent class. Such a configuration provides a convenient way to specify configuration common to the child classes of a particular class. *Class* configuration can be used to specify configuration properties for a particular class as well as the default configuration values for all methods of the class. *Overridden method* configuration allows an overriding method in a subclass to inherit the configuration of the overridden method from the superclass. *Method* configuration enables the configuration of a specific method of a given class.

Fig. C-2 shows the use of class-level annotations to tag a class as a mobile object class and to specify that public and protected methods of the class should be proxied. Furthermore, a method-level annotation is used to disable proxying of a particular method, thus effectively overriding the default class configuration. The annotation approach is considered superior to the approach adopted in JavaParty [134], in which a non-standard Java keyword is used to specify a class as mobile, thereby preventing the original application from being compiled using standard Java compilers. This approach is also favoured over the XDoclet-style configuration (as used in [35]), which although having an advantage of not requiring native sup-

port from the Java compiler since configuration properties are embedded in comment blocks, is not accessible at byte-code level (e.g. via Java reflection API).

```
1 <app>
2   <package name="java.awt">
3     <class name="Window">
4       <method name="addWindowListener" proxy="true">
5         <formal-argument type="java.awt.event.WindowListener"/>
6       </method>
7     </class>
8   </package>
9 </app>
```

Fig. C-3. Configuration using XML Files

Lastly, *deployment descriptors* are used to specify configuration properties for the pre-compilation of a specific application. This handler has the highest priority since it is the most specific and flexible configuration approach, allowing different descriptors/files to be used for different pre-compilations and even different applications. Moreover, such configuration addresses a limitation of embedded configuration (e.g. annotations) by allowing the configuration of specific classes/methods even in the absence of source code (e.g. classes belonging to external libraries). However, the drawback is that unlike annotations, modification of application code (e.g. changing method signatures) might result in configuration inaccuracy leading to inconsistency/confusion. Furthermore, such configuration is more verbose than annotations, as shown in Fig. C-3, in which an XML file is used to enable proxying for a specific method of a system class.

Transformation Manager

As mentioned in Appendix B, the Transformation Manager uses a solution based upon the strategy design pattern, whereby different strategies are applied for transforming different types of classes (as configured by the deployer). The four class types relevant to this work include: *mobile*, *stationary*, *client*, and *normal* as illustrated in Fig. C-4. Annotations are used to assign specific types to classes, with the exception of the *normal* type, which is the default class type (i.e. applies to non-annotated classes).

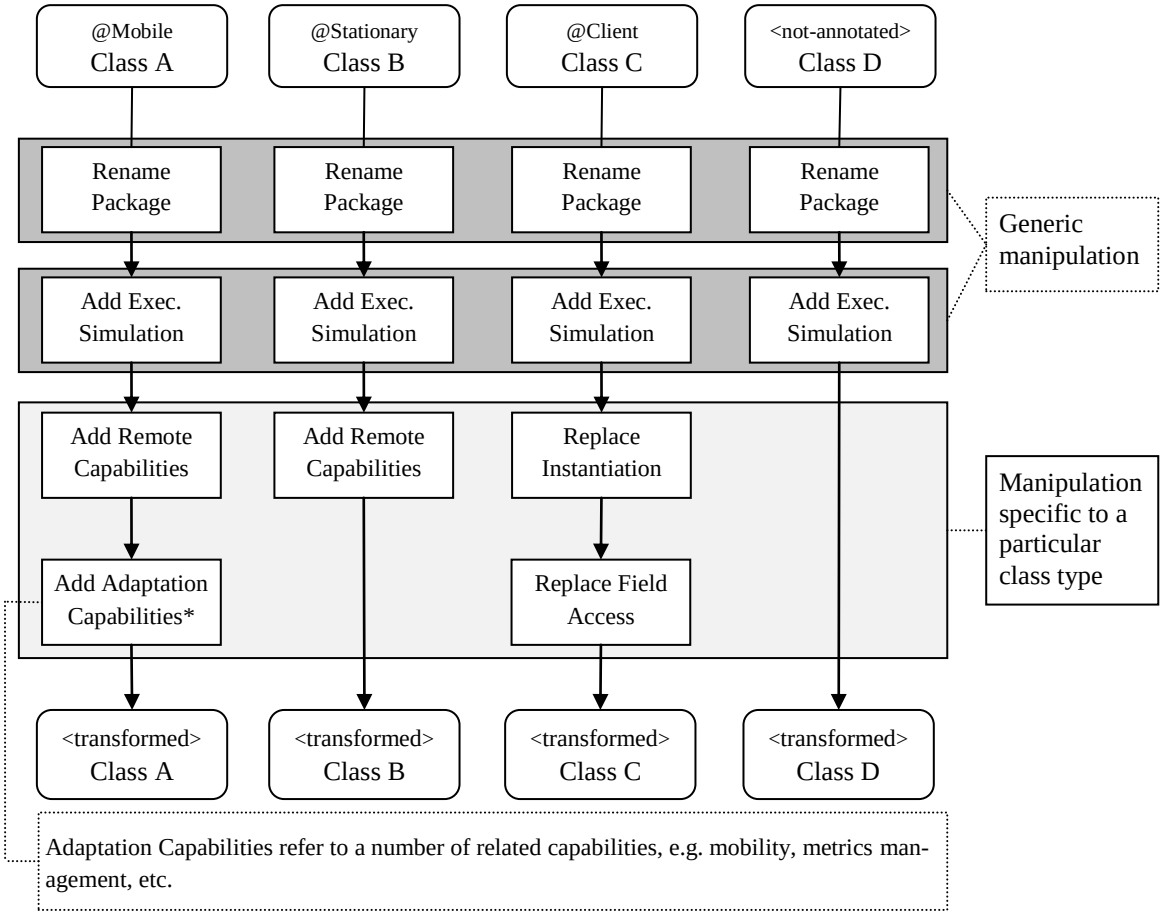

Fig. C-4. Class/Transformation Types in MobJeX

The *mobile* type is assigned to classes whose instances can migrate between hosts, thereby involving injection of capabilities such as remoteness (e.g. remote invocation, data marshalling/unmarshalling, etc.), mobility (e.g. object migration, location tracking, etc.), and adaptation (e.g. metrics management). As shown in Fig. C-4, capabilities related to mobility and adaptation are combined into a single manipulation task, which is a design decision made based on trade-offs between the development complexity involved in the separation of cohesive capabilities and the modularity/reusability of the manipulation task. Note that in this case, individual capabilities (e.g. metrics management) can still be disabled although not as easily as removing a manipulation task from the list.

Since stationary objects are accessible remotely, the injection tasks applied to *stationary* classes are largely the same as mobile classes, with the primary difference being stationary classes do not require mobility and adaptation. Consequently, the "Add Remote Capabilities" task which is common to both class types can be reused. Note that the task involves the generation of multiple proxy-related classes (e.g. the domain class) which for brevity are not explicitly shown in Fig. C-4.

The *client* type is assigned to classes whose instances potentially instantiate or access the fields of proxied objects (e.g. mobile or stationary objects). The assignment of such a class type serves to instruct the pre-compiler about classes that require specific transformation for addressing the proxy transparency issues discussed in section 6.2.4. Although not strictly needed since code analysis can be used to automatically identify code requiring such transformation, this serves as a performance safe-guard to anticipate the pre-compilation of applications containing large code bases. In this case, traversing all classes and analysing the content (e.g. method bodies) of the classes might present an unacceptable overhead, despite performance not usually being such a major concern in offline transformation.

Finally, *normal* classes refer to classes, which do not require type-specific transformation in the way that mobile, stationary, and client classes do, but may still involve generic transformation, which is applicable across all class types. Examples of generic transformation include 1) renaming the package of a class, which essentially moves the class to a different package, and 2) inserting code into a method for the purpose of simulating execution with particular intensity. The former serves to avoid naming conflicts between different versions of the same class, which may occur due to the existence of multiple versions of the same application (e.g. injected with different sets of capabilities). The latter is useful for experimenting with adaptation behaviour since it enables analysing the impact of varying application characteristics (i.e. method execution intensity or duration) on adaptation decision making, as applied in the experiments presented in Chapter 5.

Code Generator

The Code Generator is organised into multiple Java classes, each of which is responsible for generating a specific code fragment (e.g. a constructor, a method, or a statement). The generation of a code fragment uses a template (scripted in the StringTemplate language), which itself might use one or more sub-templates to allow better readability and promote re-use by other templates. An example is a sub-template for generating method declarations (e.g. name, parameter list, return type), which can be used by any template that dynamically generates methods, such as proxy methods, implementation methods, or domain methods.

Class Manager

Class information is acquired by the Class Manager via the Java reflection API, which allows a class reference (i.e. an instance of `java.lang.Class`) to be loaded dynamically using the corresponding class name, thereby allowing information to be obtained from the class reference. This requires the supplied class name to be in a *qualified* format (i.e. prefixed with the name of the package of the class) in order to remove ambiguity caused by classes having the same name but being located in different packages, thereby requiring conversion from un-

qualified class names (e.g. those acquired from JavaML representations) to the qualified versions. In addition, although qualified class names can be used to load the majority of classes, *special classes* such as those representing primitive types (e.g. integer, boolean), array types, and inner classes, are either not loadable (by the default Java class loader) or have to be loaded using specific names known as *binary names*. Such an issue is Java-specific and therefore is more appropriately addressed in the Class Manager rather than in the adopted intermediate representation.

Consequently, the Class Manager uses a custom class loader, which has the ability to translate class/type names (including unqualified names) into qualified *binary names* based on the current transformation scope or context. Qualified class names are determined from the unqualified versions by taking into account Java class import behaviour, in which classes are either *explicitly* imported using the "`import`" keyword or *implicitly* imported due to being located in the same package. This import information is relayed from XSLT during the transformation of the relevant class through the System Facade. On the other hand, *special classes* are identified by matching certain patterns or text on a given class name, and loaded either by hard-coding a fixed set of primitive types or by translating the textual name into the binary name (using specific rules according to the Java specification).

The custom class loader also addresses an issue caused by differing Java APIs used for the pre-compilation and the pre-compiler as introduced in Appendix B. In this case, a custom class loader is required to obtain a different version of system classes (for pre-compilation purposes), which presents complexity in that manual loading of system classes is not allowed by the JVM due to security concerns. As such, a technique which involves extracting class information from class byte code using a byte-code manipulation library such as ASM [28], is used. Consequently, wrapper objects are used to allow a consistent interface for accessing system and non-system class information, which differs in that the former is obtained using a byte-code manipulation library, whereas the latter is acquired using class loading for reasons of efficiency.

Post-deploy-time Transformer

In addition to the pre-compilation addressed so far in this chapter, post-deploy-time transformation is discussed in this section due to its role in addressing the heterogeneity of execution environments (i.e. Java installations/APIs) as described in section 7.1.3. This type of transformation, which is performed at the byte-code level, should be kept to a minimum, primarily due to its complexity, and therefore is applied only for specific purposes not achievable via pre-compilation. In particular, such transformation is most useful for solutions requiring modifications of the parent classes of a proxy class (e.g. modifying method modifiers), which

are the focus of section 6.2.2, because such classes may belong to a specific Java API, thereby requiring the transformation to be performed on the correct API/class version.

On the other hand, although the transformation concerning implementation classes presented in section 6.2.3 may also apply to parent classes, post-deploy-time transformation is not required in this case, since implementation classes can never be part of external libraries, due to the constraint mentioned in section 6.1.1, in which external/library classes should not be proxied. In contrast, such transformation might be required for the client code modification proposed in section 6.2.4 for the rare occasions where reflective operations are used to access a proxied object/class, because unlike regular field access and instantiation, reflective operations are resolved dynamically, thereby allowing execution on classes developed independently (of the application), e.g. library classes.

In order to facilitate automation, specific information about the pre-compiled application (e.g. classes, methods) is automatically generated by the pre-compiler (i.e. during pre-compilation) in the form of an executable Java class containing necessary transformation logic, which is to be executed in the target machine prior to application execution. The execution of the class requires certain byte-code transformation support provided by the runtime component of MobJeX (i.e. using ASM), as opposed to the compile-time component (i.e. Mobjexc), since Mobjexc does not get deployed to the target machines for reasons of efficiency. The exact phase in application deployment, in which post-deploy transformation is executed is discussed in Appendix D.

Appendix D. Deployment Process

This section presents a deployment process to demonstrate how the code transformation solution discussed in Appendix B and Appendix C, can be applied in practical scenarios to facilitate the automation of application deployment in an adaptation framework (such as MobJeX). The process, which includes the task of injecting adaptation capabilities into the deployed application, also indirectly facilitates the preceding application development since the applied capability injection alleviates the need for additional specialised development effort (in this case adaptive application partitioning via object mobility).

In practice however, the development of the application might *not* be completely independent from the capabilities provided/supported by the framework since the design/implementation could be tailored in order to take advantage of the injected capabilities. As an example, the development of applications which will utilise the injected application partitioning capability should prioritise code modularity since this allows objects to be distributed at a finer granularity, thus possibly resulting in more optimal partitioning. Nevertheless, despite being potentially affected by the supported capabilities, the development process itself is generally independent from the framework, i.e. not explicitly using the provided functionality (e.g. through an API).

The deployment process presented in Fig. D-1 consists of distinct steps, each of which groups one or more tasks with similar characteristics in terms of: 1) whether the tasks can be automated or should be executed manually by the application deployer, and 2) the scenarios in which these tasks need to be re-executed/repeated, which generally occurs because the information provided in the previous deployment/iteration (e.g. specifying adaptation thresholds, determining mobile objects/classes) was wrong or non-optimal (i.e. not favourable for the specific application or execution environment).

Repeating a specific step in the process might also (though not necessarily) require the re-execution of subsequent steps as will be described further in the discussion of the relevant steps. Note that even though certain steps can be automated (as depicted in Fig. D-1 as *auto* as opposed to *manual*), human intervention is still required to initiate the automatic execution. However, the execution of consecutive automated tasks (e.g. tasks 3-4) can be grouped so that only a single initiation step is required, although this is only possible in the ideal case where the initial configuration is correct and does not need to be revisited.

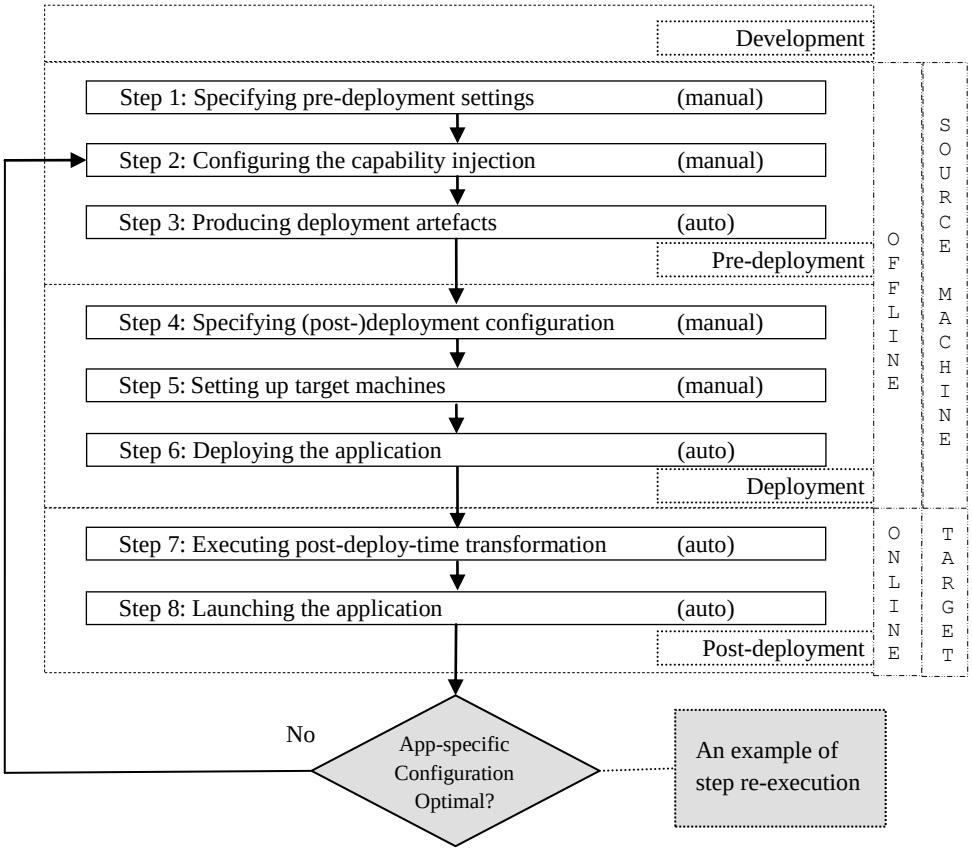

Fig. D-1. Application Deployment Process

In the proposed deployment model, the execution of certain deployment steps/tasks is facilitated by the framework via two discrete mechanisms, i.e. *offline* and *online* support, as illustrated in Fig. D-1. The distinction between offline and online support is made so that the appropriate support code can be independently installed on a machine, depending on which automated steps will be executed on the machine. In MobJeX, the online support is mainly implemented as a set of middleware containers such as services or runtimes, as discussed in section 4.3, whereas the offline support is implemented entirely in the Mobjexc pre-compiler with the exception of the Post-deploy Transformer described in Appendix C.

Post-deploy transformation is included in online support despite being considered as offline transformation in section 7.1.3, so that the distribution of support code to the target machines can be minimised by only requiring the online support to be installed in the target machines. Such a decision is particularly important when deploying a distributed/mobile application to heterogeneous target machines because it is possible that the target machines are restricted in terms of hardware resources.

On the other hand, the offline support is required only in a single pre-determined source machine since it produces cross-platform deployment artefacts (e.g. Java classes) and thus enables the deployment/distribution of the produced artefacts to heterogeneous target ma-

chines. Ideally, the source machine should be sufficiently powerful to handle the injection of the supported adaptation capabilities into the deployed application since this generally requires complex code transformation.

The following discussion describes each of the steps listed in Fig. D-1 in detail. For clarity, the deployment steps are grouped into several phases: *pre-deployment*, *deployment*, and *post-deployment*. The automation of tasks involved in the pre-deployment and deployment phases are facilitated by offline support, whereas automated post-deployment tasks are facilitated by online support.

Step 1 refers to the tasks performed in the initial stage of the deployment which involves the specification of general settings required for the execution of subsequent steps in the pre-deployment phase (steps 2-4). In this step, the application deployer specifies the paths to the files containing some of the configuration information that will be specified in step 2. Additionally, information required for steps 3 and 4, which includes the path to the application code, Java home path, library paths, output paths for generated artefacts/files, etc. should also be provided.

Step 2 involves configuring the injection of the adaptation capabilities based on specific knowledge of the deployed application, for the purpose of producing code that is of better quality (e.g. more efficient execution) than the default configuration. This step might be revisited later for fine-tuning purposes, such as explicitly specifying methods to be proxied to reduce overhead (as described in section 6.3) or enabling/disabling certain capabilities depending on the behaviour exhibited in the initial/previous execution of the application.

Step 3 involves generating deployment artefacts which include: 1) application source code resulting from the transformation presented in section 7.2, 2) *deployment scripts* consisting of execution instructions for deploying/launching the application, and 3) *configuration files* containing default configuration information for deployment/execution, such as paths to the installed JDKs. Although this step should ideally be fully automated, human involvement might be required to customise the generated code.

Step 4 addresses the configurability of subsequent tasks in the deployment and post-deployment phases through manual modification of the *deployment scripts* and *configuration files* generated in step 3. When customisation is required, the aforementioned deployment scripts are usually only modified once, whereas since the configuration files contain information about runtime environments, different versions of the files are required for heterogeneous target machines. Although in ideal cases, this step only needs to be executed once (on the source machine), the re-specification of configuration files (either on the source machine or the relevant target machines) is not uncommon due to non-optimal or inaccurate initial configuration.

Step 5 is concerned with setting up environments that are suitable for the execution of the to-be-deployed application on the target machines. This is achieved by firstly ensuring the availability of a compatible JRE as well as the online support code on the target device. Middleware containers (e.g. MobJeX *services*) are launched to provide certain functionality required for deploying and executing the application as will be described in steps 6 and 8 respectively. This step is only re-executed in subsequent deployment iterations when a different JRE version or online support installation is required, or certain middleware containers need to be restarted.

Step 6 involves compiling transformed source code into byte code and deploying the produced code, library code, and resource files to the target machines based on the configuration provided in step 4 and as such, the re-execution of step 4 also results in the re-execution of this step. The tasks performed in steps 4 and 5 enable a completely automated deployment of code and resource files, because firstly, the location (e.g. hostname or IP address) of individual target machines, which was specified in step 4, allows automatic discovery of the target machines. Secondly, the middleware containers launched in step 5, allow the source machine to contact the middleware containers and automatically distribute the application/library code to individual target machines.

Step 7 involves the execution of post-deploy transformation on specific class byte code (e.g. system classes) as described in Appendix C. This step needs to be re-executed on a specific target machine *only* if a new JRE is installed or new libraries are deployed to the machine.

Step 8 involves launching the deployed application using the generated or potentially customised *deployment scripts* and *configuration files* (described in steps 3 and 4). Depending on the application, this step might only be required for a single machine, in contrast to the post-deploy transformation (step 7) which needs to be executed on all target machines. For example, in the case of application partitioning, the entire application may initially reside in a single machine and then be partially offloaded to other machines during execution. Execution configuration (e.g. specifying application arguments, determining adaptation thresholds) could be specified in this step instead of earlier in step 4, particularly during the later fine-tuning phase where complete re-deployment is not necessary. Ideally, this step can be fully automated, since the deployed application should be able to adapt (e.g. using the adaptation solution proposed in this thesis) without human intervention.

Appendix E. Example of Transformed Code

This section provides examples of the classes produced by the transformation of a specific class (from the virtual-world application described in section 7.3.2), according to the class structure adopted in the *domain approach* presented in section 6.2.1.3. The transformation was completely automated using Mobjexc (which implements the solution described in Appendix B, as discussed in Appendix C). These examples serve to demonstrate how the capabilities of interest to this work (e.g. object mobility, metrics management), can be injected using the proxy solution proposed in section 6.2. Furthermore, they also serve to demonstrate the flexibility of the code transformation solution presented in section 7.2 by showing the complexity of the transformation addressed in this work, which involves a variety of transformation tasks such as generating new code artefacts, modifying statements, modifying method/class signatures, inserting new statements, inserting additional methods, etc.

Fig. E-1 presents an example of an original/untransformed class of the virtual world application. A small amount of unused code (the self reference in line 4) was added to illustrate a more fine-grained code modification as shown in the resulting transformation (of the implementation class) in Fig. E-6. As can be seen at line 1, the class is specified as mobile (also known as mobject) using a Java annotation, thereby causing proxy-related classes (e.g. domain class, proxy class) to be generated. The same configuration technique is also used to specify that only public methods of the original class require the corresponding methods in the proxy class as shown at line 2. Similarly, line 15 specifies that metrics collection/management should be disabled for the method `getRegion`.

```
1  @Mobject
2  @DefaultProxied( { Visibility.PUBLIC })
3  public class WorldModel extends CompositeModel {
4    protected CompositeModel self = this;
5    public WorldModel() {
6      super(new HashMap());
7      ...  // Constructor body
8    }
9    public static WorldModel getInstance() {
10     ...  // Method body
11   }
12   public void addRegion(RegionModel r) {
13     ...  // Method body
14   }
15   @Collectable(false)
16   public RegionModel getRegion(int id) {
17     ...  // Method body
18   }
19   ...  // Other declared instance methods
20 }
```

Fig. E-1. Original Class

```
1  public abstract class WorldModel extends
2      CompositeModel_MobjexImpl implements Domain {
3    protected WorldModel(java.util.Map arg0) {
4      super(arg0);
5    }
6    protected WorldModel() {
7      super();
8    }
9    protected WorldModel(CreationInfo info) {
10     super(info);
11   }
12   public static WorldModel getInstance() {
13     ...  // Original method body
14   }
15   public abstract void addRegion(RegionModel r);
16   public abstract RegionModel getRegion(int id);
17   ...  // Other declared instance methods
18 }
```

Fig. E-2. Produced Domain Class

```
1 public interface WorldModel_MobjexRemote
2     extends MobileRemote {
3   Map addRegion_MobjexRemote(
4     ProxyMethodMetricsContainer p,
5       RegionModel r) throws RemoteException;
6   Map getRegion_MobjexRemote(
7     ProxyMethodMetricsContainer p, int id)
8       throws RemoteException;
9    ...  // Other method declarations
10}
```

Fig. E-3. Produced Remote Interface

```
1 public interface WorldModel_MobjexLocal {
2   void addRegion_MobjexLocal(
3     ProxyMethodMetricsContainer p, RegionModel r);
4   RegionModel getRegion_MobjexLocal(
5     ProxyMethodMetricsContainer p, int id);
6   ...  // Other method declarations
7 }
```

Fig. E-4. Produced Local Interface

As discussed in section 6.2, in the domain approach, the transformation of a proxied class produces three main classes, namely *domain class*, *proxy class*, and *implementation class* as shown in Fig. E-2, Fig. E-5, and Fig. E-6. However, in the provided examples, two additional interfaces (i.e. *remote interface* and *local interface*), are generated due to the requirement imposed by the supported capabilities. Both the remote and local interfaces specify methods that need to be implemented by the implementation class. The remote interface (Fig. E-3) declares *remote methods*, which are invoked (via an RMI stub) from the proxy when the caller object is a different machine from the proxied/mobile object. On the other hand, *local methods*, which are specified in the local interface (Fig. E-4), are directly called (without using RMI) during local invocation for reasons of efficiency. Note that unique name suffixes (e.g. `_MobjexRemote`) are used for generated classes, interfaces and methods in order to reduce the probability of conflicts. In MobJeX, these prefixes are customisable, meaning that in the event of a conflict, they can be modified through configuration.

As can be seen in Fig. E-5 (i.e. proxy class) and Fig. E-6 (i.e. implementation class), code generated for individual methods is similar, and thus it is possible to generalise the code to allow it to be shared by multiple methods. However, this introduces runtime performance overhead as it requires the use of Java reflection for dynamically resolving methods and classes. Furthermore, in such an approach, additional conditional statements need to be inserted in order to enable/disable certain capabilities for certain methods since it cannot be assumed that all methods support the same sets of capabilities. In contrast, in the adopted approach, if certain capabilities are to be disabled for a specific method, the relevant code is simply excluded from the injection. As such, support code which primarily delegates to the core functionality implemented in the MobJeX framework/middleware, is inserted into each method, e.g. lines 19-46 of Fig. E-5.

The following discusses the injection of *object mobility*, *error handling*, *metrics management*, and *synchronisation* capabilities, each of which primarily involves the insertion of cohesive sub-capabilities in *proxy methods* (of the proxy class) and/or *local/remote methods* (of the implementation class). Some capabilities might require the insertion of new fields or methods however it is beyond the scope of this thesis to provide an exhaustive examination of different capabilities since the focus is on code transformation rather than the specific

techniques/strategies adopted by MobJeX. Nonetheless, a brief discussion of some of these techniques or strategies is provided to demonstrate the flexibility of the structure of the generated classes.

The injection of *object mobility* requires the insertion of two major sub-capabilities, which include handling method invocation and tracking mobile-object location. The rest of the functionality including the actual migration logic is implemented in the framework/middleware.

Handling method invocation involves forwarding the invocation to the appropriate object depending on the locality of the client/proxy and the implementation/mobile object as shown at lines 22-29 of the proxy class (Fig. E-5). Furthermore, upon returning from the remote method call, the implementation object should also return parameters that have potentially been modified as shown at lines 37-38 (Fig. E-6). Returning the parameter allows copying its possibly modified properties back to the original parameter as shown at line 28 of the proxy class (Fig. E-5) in order to emulate pass-by-reference behaviour which is consistent with the local invocation. Note that this operation is not required/injected for arguments that are known to be immutable, e.g. primitive values, as is the case of the argument of `getRegion()` (line 48, Fig. E-5).

```
1 public class WorldModel_MobjexProxy extends WorldModel implements Proxy, ... {
2    private CompositeModel_MobjexProxy parentProxy;
3    private ProxyDelegate proxyDelegate;
4    public WorldModel_MobjexProxy(CreationInfo mobjexCreationInfo) {
5       super(mobjexCreationInfo);
6       this.parentProxy = new CompositeModel_MobjexProxy(mobjexCreationInfo);
7       this.proxyDelegate = this.parentProxy.getProxyDelegate_Mobjex();
8       ...  // Initialisation of metrics containers
9
10   }
11   public final boolean referenceEquals_Mobjex(Object obj) {
12      return this.parentProxy.referenceEquals_Mobjex(obj);
13   }
14   public final void setFieldValue_Mobjex(Field field, Object value) throws ... {
15      Method setter = this.proxyDelegate.getFieldSetter(field);
16      setter.invoke(this, new Object[] { value});
17   }
18   public void addRegion(RegionModel arg_r) {
19      do {
20         try {
21            getMetricsContainer_Mobjex(1).startMetricsCollection();
22            if (this.proxyDelegate.isLocal()) {
23               ((WorldModel_MobjexLocal) getMobject()).addRegion_MobjexLocal(
24                     getMetricsContainer_Mobjex(1), arg_r);
25            } else {
26               Map m = ((WorldModel_MobjexRemote) getStub()).addRegion_MobjexRemote(
27                     getMetricsContainer_Mobjex(1), arg_r);
28               Duplicator.shallowCopyProperties(m.get("arg_r"), arg_r);
29            }
30            getMetricsContainer_Mobjex(1).endMetricsCollection();
31            this.proxyDelegate.resetErrorInfo();
32            return;
33         } catch (MovedException e) {
34            this.proxyDelegate.updateObjectLocation();
35            addRegion(arg_r);
36            return;
37         } catch (RemoteException e) {
38            if (!this.proxyDelegate.handleError(e)) {
39               throw new ProxyException("Could not connect to remote object", e);
40            }
41         } catch (MobjexLockDiscardedException e) {
42            this.proxyDelegate.updateObjectLocation();
43            addRegion(arg_r);
44            return;
45         }
46      } while (true);
47   }
48   public RegionModel getRegion(int arg_id) {
49      do {
50         try {  // Metrics collection disabled for this method
51            RegionModel r;
52            if (this.proxyDelegate.isLocal()) {
53               r = ((WorldModel_MobjexLocal) getMobject()).getRegion_MobjexLocal(null, arg_id);
54            } else {
55               Map m = ((WorldModel_MobjexRemote)getStub()).getRegion_MobjexRemote(null, arg_id);
56               r = ((virtualworld.model.regionmodel.RegionModel) m.get("retValue"));
57            }
58            this.proxyDelegate.resetErrorInfo();
59            return r;
60         }
61         ...
62      } while (true);
63   }
64   public void setMobjexField_virtualworld_model_worldmodel_WorldModel_self(CompositeModel arg_val) {
65      ...
66   }
67   public virtualworld.model.AbstractModel get(java.lang.Object arg_arg0) {
68      return this.parentProxy.get(arg_arg0);
69   }
70   ...  // Other methods
71}
```

Fig. E-5. Produced Proxy Class

Tracking the location of a mobile object requires a notification (in the form of a Java exception) to be sent to the proxy when it attempts to access an object that has moved. Consequently, the proxy has to catch the exception and acquire the new location of the object from the middleware as shown at lines 33-36 in Fig. E-5. An alternative strategy would be to im-

mediately notify the proxy upon the migration of the mobile object rather than having to wait until the proxy attempts to access the mobile object. However, this alternative strategy places more burden on the middleware since it has to keep track of all the proxies of the mobile object in order to allow the notification. While it is not the intention of this thesis to provide a critical evaluation based on the relative merits of either strategy, the example is illustrative in terms of code transformation, since adopting the latter strategy requires only the insertion of an additional method to allow the middleware to notify relevant proxies.

The injection of the *error handling* capability requires the insertion of a retry loop to allow a proxy to attempt to communicate with a remote mobile object for a certain number of times before giving up (lines 19-46 in Fig. E-5). Lines 38-40 (Fig. E-5) show the insertion of the code for recovering from errors, the implementation of which is provided by the framework.

The injection of object clustering involves the insertion of code for keeping track of the current call/execution stack as shown in lines 16 and 18 of Fig. E-6. The use of this stack is based on a similar idea described in section 4.1.1.2, in which a stack is used to identify the caller of the mobile object (for the purpose of keeping track of the interaction between individual objects). In object clustering, such functionality is required to determine the cluster relationship between the caller and the callee. In the case where the caller is in a different cluster from the callee, proxied objects serving as method arguments or return values, need to be either wrapped/proxied or unwrapped/unproxied depending on certain logic or rules, the discussion of which is beyond the focus of this thesis. On the other hand, if the caller is in the same cluster as the callee, communicated proxied objects should not be modified since both the caller and the callee should access them in the same way.

```
1 public class WorldModel_MobjexImpl extends WorldModel
2       implements WorldModel_MobjexLocal, WorldModel_MobjexRemote, Mobile, ... {
3    private MobileDelegate mobileDelegate;
4    protected CompositeModel self;
5    public WorldModel_MobjexImpl() {
6       super(new HashMap());
7       initMobileDelegate_Mobjex();
8       ...  // Initialisation of metrics containers
9       this.self = (WorldModel) MobjexRuntimeImpl.getInstance().createProxyIfNecessary(this, "");
10      ...  // Original constructor body
11   }
12   protected WorldModel_MobjexImpl(CreationInfo mobjexCreationInfo) {
13      super(mobjexCreationInfo);
14   }
15   public void addRegion(RegionModel arg_r) {
16      MobjectCallStack.pushMethod(this, getMetricsContainer_Mobjex(1));
17      ...  // Original method body
18      MobjectCallStack.pop();
19   }
20   public void addRegion_MobjexLocal(ProxyMethodMetricsContainer p, RegionModel arg_r) {
21      if (this.mobileDelegate.tryExecuteLock()) {
22         try {
23            getMetricsContainer_Mobjex(1).startMetricsCollection(p, new Object[] { arg_r});
24            addRegion(arg_r);
25            getMetricsContainer_Mobjex(1).endMetricsCollection(p);
26            return;
27         } finally {
28            this.mobileDelegate.releaseExecuteLock();
29         }
30      } else {
31         throw new RuntimeException("Failed in getting execute lock on the object.");
32      }
33   }
34   public Map addRegion_MobjexRemote(ProxyMethodMetricsContainer p, RegionModel arg_r) {
35      Map results = new HashMap();
36      addRegion_MobjexLocal(p, arg_r);
37      results.put("arg_r", arg_r);
38      return results;
39   }
40   public void initMobileDelegate_Mobjex() {
41      if (this.mobileDelegate == null) {
42         super.initMobileDelegate_Mobjex();
43         this.mobileDelegate = super.getMobileDelegate_Mobjex();
44      }
45   }
46   ...  // Other methods
47}
```

Fig. E-6. Produced Implementation Class

The injection of *metrics management* involves the insertion of two core sub-capabilities, which include collecting metrics and centralising metrics storage. The injection is only required for software metrics (e.g. execution time) since system/resource metrics (e.g. processor usage) can be collected independently by the middleware.

Similar to object clustering, code inserted at lines 16 and 18 (Fig. E-6) is used to identify caller objects for the purpose of collecting interaction metrics (e.g. invocation frequency) as was addressed in section 4.1.1.2. Collecting the metrics of a method involves inserting code before and after the execution of the method into the relevant proxy method (lines 21 and 30, Fig. E-5) and local method (lines 23 and 25, Fig. E-6). Note that capabilities inserted into a local method are also executed when the corresponding remote method is invoked (i.e. during remove invocation) due to method forwarding as shown in line 36 of Fig. E-6.

Note that collected software metrics are stored in the implementation object and delivered to the adaptation engine at a later stage (e.g. during decision making) using the pull approach discussed in section 4.2.2. Consequently, proxies are required to pass an additional parameter

(containing proxy metrics) during the invocation of a local/remote method of the implementation object as shown at lines 24 and 27 of Fig. E-5.

The injection of the *synchronisation* capability involves inserting code for acquiring and releasing a lock in the local method (lines 21 and 28 in Fig. E-6) to allow a mobile object to migrate while its method is executing without corrupting the state of the mobile object. Furthermore, synchronisation allows a proxy to receive a notification (via a Java exception) of a migration that is in progress (lines 41-44 in Fig. E-5). The notification is useful for improving the parallelism of the application execution since it allows the proxy to execute certain operations such as locating the target destination of the mobile object, to anticipate the migration before it completes.

References

[1] E. Abebe and C. Ryan, "A Distributed Approach to Local Adaptation Decision Making for Sequential Applications in Pervasive Environments," in *Proceedings of the Confederated International Conferences, CoopIS, DOA, IS, and ODBASE 2009*, Vilamoura, Portugal, 2009, pp. 773-789.

[2] G. D. Abowd, C. G. Atkeson, J. Hong, S. Long, R. Kooper, and M. Pinkerton, "Cyberguide: A Mobile Context-Aware Tour Guide," *Wireless Networks,* vol. 3, pp. 421-433, October 1997.

[3] A. Acharya, M. Ranganathan, and J. Saltz, "Sumatra: A Language for Resource-Aware Mobile Programs," in *Mobile Object Systems: Towards the Programmable Internet*, Germany, 1997, pp. 111-130.

[4] V. Akman and M. Surav, "The Use of Situation Theory In Context Modeling," *Computational Intelligence,* vol. 13, pp. 427-438, December 17 1997.

[5] B. Aktemur, J. Jones, S. Kamin, and L. Clausen, "Optimizing Marshalling by Run-Time Program Generation," in *Proceedings of 4th International Conference on Generative Programming and Component Engineering (GPCE)*, Tallinn, Estonia, 2005, pp. 221-236.

[6] P. Anderson, "The Performance Penalty of XML for Program Intermediate Representations," in *Fifth IEEE International Workshop on Source Code Analysis and Manipulation*, 2005, pp. 193-201.

[7] Antlr, "ANTLR Parser Generator: http://www.antlr.org/," 2009.

[8] Y. Aridor, M. Factor, and A. Teperman, "cJVM: A Single System Image of a JVM on a Cluster," in *Proceedings of the International Conference on Parallel Processing*, Wakamatsu, Japan, 1999, pp. 4-11.

[9] E. Arisholm, L. C. Briand, and A. Foyen, "Dynamic Coupling Measurement for Object-Oriented Software," *IEEE Transactions on Software Engineering (TSE),* vol. 30, pp. 491-506, 2004.

[10] AspectJ, "The AspectJ Project," 2010. http://www.eclipse.org/aspectj/

[11] B. Aziz and C. Jensen, "Adaptability in CORBA: The Mobile Proxy Approach," in *Proceedings of International Symposium on Distributed Objects and Applications*, Antwerp, Belgium, 2000, pp. 295-304.

[12] G. J. Badros, "JavaML: A Markup Language for Java Source Code," *Computer Networks (Amsterdam, Netherlands: 1999),* vol. 33, pp. 159-177, 2000.

[13] S. Baek, H. Lee, S. Lim, and J. Huh, "Managing Mechanism for Service Compatibility and Interaction Issues in Context-Aware Ubiquitous Home," *IEEE Transactions on Consumer Electronics,* vol. 51, pp. 524-528, May 2005.

[14] E. Bainomugisha, J. Vallejos, É. Tanter, E. G. Boix, P. Costanza, W. D. Meuter, and T. D'Hondt, "Resilient actors: a runtime partitioning model for pervasive computing services," in *Proceedings of the 2009 international conference on Pervasive services*, London, United Kingdom, 2009, pp. 31-40.

[15] J. Balasubramanian, D. C. Schmidt, L. Dowdy, and O. Othman, "Evaluating the Performance of Middleware Load Balancing Strategies," in *Proceedings of the Eighth IEEE International Enterprise Distributed Object Computing Conference*, Monterey, California, 2004, pp. 135-146.

[16] M. Baldauf, S. Dustdar, and F. Rosenberg, "A Survey on Context-Aware Systems," *International Journal of Ad Hoc and Ubiquitous Computing,* vol. 2, pp. 263 - 277, 2007.

[17] F. Baude, D. Caromel, F. Huet, L. Mestre, and J. Vayssiere, "Interactive and Descriptor-Based Deployment of Object-Oriented Grid Applications," in *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, Edinburgh, Scotland, 2002, pp. 93-102.

[18] I. D. Baxter, "Design Maintenance Systems," *Communications of the ACM,* vol. 35, pp. 73-89, April 1992.

[19] T. Berners-Lee, J. Hendler, and O. Lassila, "The Semantic Web," *Scientific American,* vol. 284, pp. 34-43, 2001.

[20] L. Bettini and R. D. Nicola, "Translating Strong Mobility into Weak Mobility," in *Proceedings of the 5th International Conference on Mobile Agents* Atlanta, GA, USA, 2001.

[21] V. Bharghavan and V. Gupta, "A Framework for Application Adaptation in Mobile Computing Environments," in *Proceedings of the 21st International Computer Software and Applications Conference*, Washington, DC, USA, 1997, pp. 573-579.

[22] D. Binkley, M. Ceccato, M. Harman, F. Ricca, and P. Tonella, "Automated Refactoring of Object Oriented Code into Aspects," in *21st IEEE International Conference on Software Maintenance*, Budapest, 2005, pp. 27-36.

[23] G. S. Blair, G. Coulson, L. Blair, H. Duran-Limon, P. Grace, R. Moreira, and N. Parlavantzas, "Reflection, Self-Awareness and Self-Healing in OpenORB," in *Proceedings of the first workshop on Self-healing Systems*, 2002.

[24] J. Bolliger and T. Gross, "A Framework-Based Approach to the Development of Network-Aware Applications," *IEEE Transactions on Software Engineering,* vol. 24, pp. 376-390 May 1998.

[25] B. Bouzy and T. Cazenave, "Using the Object Oriented Paradigm to Model Context in Computer Go," in *Proceedings of the First International and Interdisciplinary Conference on Modeling and Using Context*, Rio de Janeiro, Brazil, 1997.

[26] M. G. J. v. d. Brand, A. v. Deursen, J. Heering, H. A. d. Jong, M. d. Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser, and J. Visser, "The ASF+SDF Meta-Environment: A Component-Based Language Development Environment," in *Proceedings of the 10th International Conference on Compiler Construction*, Genova, Italy, 2001, pp. 365-370.

[27] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser, "Stratego/XT 0.16: Components for Transformation Systems," in *Proceedings of the 2006 ACM SIGPLAN symposium on Partial Evaluation and Semantics-based Program Manipulation*, Charleston, South Carolina, 2006.

[28] E. Bruneton, R. Lenglet, and T. Coupaye, "ASM: A Code Manipulation Tool to Implement Adaptable Systems," in *Adaptable and Extensible Component Systems*, Grenoble, France, 2002.

[29] H. Byun and K. Cheverst, "Utilizing Context History to Provide Dynamic Adaptations," *Applied Artificial Intelligence,* vol. 18, pp. 533-548, July 2004 2004.

[30] L. Capra, W. Emmerich, and C. Mascolo, "CARISMA: Context-Aware Reflective Middleware System for Mobile Applications," *IEEE Transactions on Software Engineering,* vol. 29, pp. 929--945, 2003.

[31] L. Capra, W. Emmerich, and C. Mascolo, "Reflective Middleware Solutions for Context-Aware Applications," in *Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, Kyoto, Japan, 2001, pp. 126-133.

[32] J. Casas, D. Clark, R. Konuru, S. Otto, R. Prouty, and J. Walpole, "MPVM: A Migration Transparent Version of PVM," *Computing Systems,* vol. 8, pp. 171-216, 1995.

[33] J. Casas, R. Konuru, S. W. Otto, R. Prouty, and J. Walpole, "Adaptive Load Migration Systems for PVM," in *Proceedings of the 1994 Conference on Supercomputing*, Manchester, England, 1994, pp. 390-399.

[34] S. J. Caughey and S. K. Shrivastava, "Architectural Support for Mobile Objects in Large Scale Distributed Systems," in *Fourth International Workshop on Object-Orientation in Operating Systems*, Lund, 1995, pp. 38-47.

[35] V. Cepa and M. Mezini, "MobCon: A Generative Middleware Framework for Java Mobile Applications," in *Proc. of Hawaii International Conference on System Sciences (HICSS-38)*, Hawaii, 2005, pp. 283b- 283b.

[36] A. T. S. Chan and S.-N. Chuang, "MobiPADS: A Reflective Middleware for Context-Aware Mobile Computing " *IEEE Transactions on Software Engineering,* vol. 29, pp. 1072-1085, 2003.

[37] F. Chang and V. Karamcheti, "Automatic Configuration and Run-time Adaptation of Distributed Applications," in *Proceedings of the 9th IEEE International Symposium on High Performance Distributed Computing*, Pittsburgh, Pennsylvania, 2000, pp. 11-20.

[38] G. Chen, B.-T. Kang, M. T. Kandemir, N. Vijaykrishnan, M. J. Irwin, and R. Chandramouli, "Studying Energy Trade Offs in Offloading Computation/Compilation in Java-Enabled Mobile Devices," *IEEE Transactions on Parallel and Distributed Systems,* vol. 15, pp. 795-809, 2004.

[39] H. Chen, T. Finin, and A. Joshi, "An Ontology for Context-Aware Pervasive Computing Environments," *The Knowledge Engineering Review,* vol. 18, pp. 197-207, September 2003.

[40] J.-L. Chen, F.-J. Wang, and Y.-L. Chen, "Program Slicing: An Application of Object-oriented Program Dependency Graphs," in *Proceedings of the Technology of Object-Oriented Languages and Systems*, Beijing, China, 1997, p. 121.

[41] K. Cheverst, N. Davies, K. Mitchell, A. Friday, and C. Efstratiou, "Developing a Context-Aware Electronic Tourist Guide: Some Issues and Experiences," in *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, The Hague, The Netherlands, 2000, pp. 17-24.

[42] G. Cohen, J. Chase, and D. Kaminsky, "Automatic Program Transformation with JOIE," in *1998 USENIX Annual Technical Symposium*, 1998, pp. 167-178.

[43] J. R. Cordy, "TXL - A Language for Programming Language Tools and Applications," in *Proceedings of the Fourth Workshop on Language Descriptions, Tools, and Applications (LDTA 2004)*, 2004, pp. 3-31.

[44] F. M. David, B. Donkervoet, J. C. Carlyle, E. M. Chan, and R. H. Campbell, "Supporting Adaptive Application Mobility," in *On the Move to Meaningful Internet Systems 2007: OTM Workshops*, 2007, pp. 896-905.

[45] A. K. Dey, G. D. Abowd, and D. Salber, "A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications," *Human-Computer Interaction,* vol. 16, pp. 97-166, December 2001.

[46] R. E. Diaconescu, L. Wang, Z. Mouri, and M. Chu, "A Compiler and Runtime Infrastructure for Automatic Program Distribution," in *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*, Denver, Colorado, 2005, p. 52a.

[47] R. Douence, P. Fradet, and M. Sudholt, "A Framework for the Detection and Resolution of Aspect Interactions," in *Proceedings of the 1st ACM SIGPLAN/SIGSOFT conference on Generative Programming and Component Engineering*, Pittsburgh, PA, USA, 2002, pp. 173-188.

[48] R. Douence, P. Fradet, and M. Südholt, "Composition, Reuse and Interaction Analysis of Stateful Aspects," in *Proceedings of the 3rd international conference on Aspect-Oriented Software Development*, Lancaster, UK, 2004, pp. 141-150.

[49] P. Eugster, "Uniform Proxies for Java," in *Conference on Object Oriented Programming Systems Languages and Applications*, Portland, Oregon, USA, 2006, pp. 139-152.

[50] J. Euzenat, J. Pierson, and F. Ramparany, "Dynamic Context Management for Pervasive Applications," *The Knowledge Engineering Review,* vol. 23, pp. 21-49, March 2008.

[51] T. Fahringer and A. Jugravu, "JavaSymphony: A New Programming Paradigm to Control and Synchronize Locality, Parallelism and Load Balancing for Parallel and Distributed Computing," *Concurrency and Computation: Practice & Experience,* vol. 17, pp. 1005 - 1025, June 2005.

[52] V. Felea and B. Toursel, "Adaptive Distributed Execution of Java Applications," in *12th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, Spain, 2004, pp. 16- 21.

[53] V. Felea and B. Toursel, "Dynamic Load-Balancing Mechanism for Distributed Java Applications," *Concurrency and Computation: Practice & Experience,* vol. 18, pp. 305-331, March 2006.

[54] N. Fenton, "Software Measurement: A Necessary Scientific Basis," *IEEE Transactions on Software Engineering,* vol. 20, pp. 199-206, March 1994.

[55] J. Floch, S. O. Hallsteinsen, E. Stav, F. Eliassen, K. Lund, and E. Gjorven, "Using Architecture Models for Runtime Adaptability," *IEEE Software,* vol. 23, pp. 62-70, 2006.

[56] S. Fu and C.-Z. Xu, "Service Migration in Distributed Virtual Machines for Adaptive Grid Computing," in *Proceedings of the 2005 International Conference on Parallel Processing*, Oslo, Norway, 2005, pp. 358-365.

[57] M. M. Fuad and M. J. Oudshoorn, "AdJava: Automatic Distribution of Java Applications," in *Proceedings of the twenty-fifth Australasian Conference on Computer science*, Melbourne, Victoria, Australia, 2002, pp. 65-75.

[58] A. Fuggetta, G. P. Picco, and G. Vigna, "Understanding Code Mobility," *IEEE Transactions on Software Engineering,* vol. 24, pp. 342-361, 1998.

[59] S. Funfrocken, "Transparent Migration of Java-Based Mobile Agents," in *Proceedings of the Second International Workshop on Mobile Agents*, Stuttgart, Germany, 1998, pp. 26-37.

[60] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software*: Addison-Wesley, 1995.

[61] H. Gani, C. Ryan, and P. Rossi, "Runtime Metrics Collection for Middleware Supported Adaptation of Mobile Applications," in *International Workshop on Adaptive and Reflective Middleware*, Melbourne, Australia, 2006, pp. 2-7.

[62] F. Garcia, M. F. Bertoa, C. Calero, A. Vallecillo, F. Ruiz, M. Piattini, and M. Genero, "Towards a Consistent Terminology for Software Measurement," *Information and Software Technology,* vol. 48, pp. 631-644, August 2006.

[63] D. Garti, S.-T. Cohen, A. Barak, A. Keren, and R. Szmit, "Object Mobility for Performance Improvements of Parallel Java Applications," *Journal of Parallel and Distributed Computing,* vol. 60, pp. 1311 - 1324, October 2000.

[64] H. Gazit, I. Ben-Shaul, and O. Holder, "Monitoring-Based Dynamic Relocation of Components in Fargo," in *Proceedings of the Second International Symposium on Agent Systems and Applications and Fourth International Symposium on Mobile Agents*, 2000, pp. 221–234.

[65] K. Geihs, M. U. Khan, R. Reichle, A. Solberg, S. Hallsteinsen, and S. Merral, "Modeling of Component-based Adaptive Distributed Applications," in *Proceedings of the 2006 ACM symposium on Applied Computing*, Dijon, France, 2006, pp. 718-722.

[66] N. Geoffray, G. Thomas, and B. Folliot, "Live and Heterogeneous Migration of Execution Environments " in *On the Move to Meaningful Internet Systems 2006: OTM Workshops*, 2006, pp. 1254-1263.

[67] N. Geoffray, G. Thomas, and B. Folliot, "Transparent and Dynamic Code Offloading for Java Applications," in *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE*, Montpellier, France, 2006, pp. 1790-1806.

[68] S. Ghita, W. Nejdl, and R. Paiu, "Semantically Rich Recommendations in Social Networks for Sharing, Exchanging and Ranking Semantic Context," in *ESWC Workshop on Ontologies in P2P Communities*, 2005, pp. 293-307.

[69] R. H. Glitho, E. Olougouna, and S. Pierre, "Mobile Agents and Their Use for Information Retrieval: A Brief Overview and an Elaborate Case Study," *IEEE Network,* vol. 16, pp. 34-41, Jan/Feb 2002.

[70] P. Grace, G. S. Blair, and S. Samuel, "Middleware Awareness in Mobile Computing," in *Proceedings of the 23rd International Conference on Distributed Computing Systems*, Providence, Rhode Island, USA, 2003, pp. 382- 387.
[71] J. D. Gradecki and J. C. John, *Mastering Apache Velocity*: Wiley & Sons Inc, 2003.
[72] P. D. Gray and D. Salber, "Modelling and Using Sensed Context Information in the Design of Interactive Applications," in *Proceedings of the 8th IFIP International Conference on Engineering for Human-Computer Interaction*, Toronto, Canada, 2001, pp. 317-336.
[73] X. Gu, A. Messer, I. Greenberg, D. Milojicic, and K. Nahrstedt, "Adaptive Offloading for Pervasive Computing " *Pervasive Computing,* vol. 3, pp. 66- 73, 2004.
[74] A. Helsinger, M. Thome, and T. Wright, "Cougaar: A Scalable, Distributed Multi-Agent Architecture," in *IEEE International Conference on Systems, Man and Cybernetics*, 2004, pp. 1910-1917.
[75] K. Henricksen and J. Indulska, "Modelling and Using Imperfect Context Information," in *Proceedings of the Second IEEE Annual Conference on Pervasive Computing and Communications Workshops*, Orlando, Florida, 2004, p. 33.
[76] K. Henricksen, J. Indulska, and A. Rakotonirainy, "Using Context and Preferences to Implement Self-Adapting Pervasive Computing Applications," *Software-practice & Experience,* vol. 36, pp. 1307-1307, August 17 2006.
[77] M. Hicks, S. Jagannathan, R. Kelsey, J. T. Moore, and C. Ungureanu, "Transparent Communication for Distributed Objects in Java," in *Proceedings of the ACM 1999 conference on Java Grande*, San Francisco, California, United States, 1999, pp. 160 - 170.
[78] J. Hielscher, R. Kazhamiakin, A. Metzger, and M. Pistore, "A Framework for Proactive Self-Adaptation of Service-Based Applications Based on Online Testing " in *Proceedings of the 1st European Conference on Towards a Service-Based Internet*, Madrid, Spain 2008, pp. 122 - 133.
[79] O. Holder, I. Ben-Shaul, and H. Gazit, "Dynamic Layout of Distributed Applications in FarGo," in *Proceedings of the 21st international conference on Software engineering*, Los Angeles, California, United States, 1999, pp. 163 -173.
[80] O. Holder, I. Ben-Shaul, and H. Gazit, "System Support for Dynamic Layout of Distributed Applications," in *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, Austin, TX, USA, 1999, pp. 403-411.
[81] J. K. Hollingsworth and P. J. Keleher, "Prediction and Adaptation in Active Harmony " *Cluster Computing,* vol. 2, pp. 195-205, October 31 2004.
[82] C. C. Holt, "Forecasting Seasonals and Trends by Exponentially Weighted Moving Averages," *International Journal of Forecasting,* vol. 20, pp. 5-10 28 January 2004.
[83] J. I. Hong and J. A. Landay, "An Infrastructure Approach to Context-Aware Computing," *Human-Computer Interaction,* vol. 16, pp. 287-303, December 2001.
[84] N. Houssos, K. Kafounis, V. Gazis, and N. Alonistioti, "Application-Transparent Adaptation in Wireless Systems Beyond 3G," *International Journal of Management and Decision Making,* vol. 6, pp. 81 - 100, 2005.
[85] R. Hull, P. Neaves, and J. Bedford-roberts, "Towards Situated Computing," in *Proceedings of the 1st IEEE International Symposium on Wearable Computers*, Cambridge, Massachusetts, USA, 1997, pp. 146-153.
[86] G. C. Hunt and M. L. Scott, "The Coign Automatic Distributed Partitioning System," in *Proceedings of the third symposium on Operating Systems Design and Implementation*, New Orleans, Louisiana, United States, 1999, pp. 187-200.
[87] C. Hütter and T. Moschny, "Runtime Locality Optimizations of Distributed Java Applications " in *Proceedings of the 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*, Toulouse, France, 2008, pp. 149-156.

[88] Java2XML, "Java2XML Project Page," 2010. https://java2xml.dev.java.net/
[89] JavaCC, "JavaCC Project Home," 2010. https://javacc.dev.java.net/
[90] JavaParty, "JavaParty - Java's Companion for Distributed Computing," 2010. http://svn.ipd.uni-karlsruhe.de/trac/javaparty/wiki/JavaParty
[91] JBoss Community, "Byteman," 2010. http://www.jboss.org/byteman
[92] F. V. Jensen, *Introduction to Bayesian Networks* vol. 1: Springer, 1997.
[93] G. Judd and P. Steenkiste, "Providing Contextual Information to Pervasive Computing Applications," in *Proceedings of the First IEEE International Conference on Pervasive Computing and Communications*, Dallas-Fort Worth, Texas, US, 2003, pp. 133-142.
[94] E. Jul, H. Levy, N. Hutchinson, and A. Black, "Fine-grained mobility in the Emerald system," *ACM Transactions on Computer Systems (TOCS),* vol. 6, pp. 109-133, February 1988.
[95] S. Kamin, L. Clausen, and A. Jarvis, "Jumbo: Run-time Code Generation for Java and its Applications," in *International Symposium on Code Generation and Optimization (CGO'03)*, San Francisco, California, 2003, pp. 48-56.
[96] R. Kapitza, H. Schmidt, G. Soldner, and F. J. Hauck, "A Framework for Adaptive Mobile Objects in Heterogeneous Environments," in *Proceedings of the International Symposium on Distributed Objects and Applications (DOA)*, Montpellier, France, 2006, pp. 1739-1756.
[97] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-Oriented Programming," in *Proceedings European Conference on Object-Oriented Programming*, Berlin, Heidelberg, and New York", 1997, pp. 220-242.
[98] Y. Kim and K. Lee, "A Quality Measurement Method of Context Information in Ubiquitous Environments," in *Proceedings of the 2006 International Conference on Hybrid Information Technology*, Jeju Island, Korea, 2006, pp. 576-581.
[99] K. Kono and T. Masuda, "Efficient RMI: Dynamic Specialization of Object Serialization," in *Proceedings of The 20th International Conference on Distributed Computing Systems (ICDCS 2000)*, Taipei, Taiwan, 2000, p. 308.
[100] P. Korpipaa, J. Mantyjarvi, J. Kela, H. Keranen, and E.-J. Malm, "Managing Context Information in Mobile Devices," *IEEE Pervasive Computing,* vol. 2, pp. 42-51, July 2003.
[101] M. Krause and I. Hochstatter, "Challenges in Modelling and Using Quality of Context (QoC)," in *Mobility Aware Technologies and Applications*, 2005, pp. 324-333.
[102] R. Krummenacher, J. Kopecky, and T. Strang, "Sharing Context Information in Semantic Spaces " in *On the move to meaningful internet systems 2005: OTM Workshops*, 2005, pp. 229-232.
[103] T. Kuhn and O. Thomann, "Abstract Syntax Tree," 2010. http://www.eclipse.org/articles/Article-JavaCodeManipulation_AST/index.html
[104] W.-H. F. Leung, "Program Entanglement, Feature Interaction and the Feature Language Extensions," *Computer Networks: The International Journal of Computer and Telecommunications Networking,* vol. 51, pp. 480-495, February 2007.
[105] C. Liang, N. C. Hock, and Z. Liren, "Feedback Control Using Representatives and Timer in Reliable Multicast Protocols," in *Proceedings of IEEE Region 10 Conference (TENCON 2000)*, Kuala Lumpur, Malaysia, 2000, pp. 338-340.
[106] H. Lieberman and T. Selker, "Out of Context: Computer Systems that Adapt to, and Learn from, Context," *IBM Systems Journal,* vol. 39, pp. 617-632, July 2000.
[107] A. Lima, W. Cirne, F. Brasileiro, and D. Fireman, "A Case for Event-Driven Distributed Objects," in *Proceedings of the International Symposium on Distributed Objects and Applications (DOA)*, Montpellier, France, 2006, pp. 1705-1721.
[108] N. Liogkas, B. MacIntyre, E. D. Mynatt, Y. Smaragdakis, E. Tilevich, and S. Voida, "Automatic Partitioning for Prototyping Ubiquitous Computing Applications," *IEEE Pervasive Computing,* vol. 3, pp. 40- 47, July-Sept 2004.

[109] H. Lufei and W. Shi, "Fractal: A Mobile Code Based Framework for Dynamic Application Protocol Adaptation in Pervasive Computing," in *Proceedings of Parallel and Distributed Processing Symposium*, 2005, p. 40a.

[110] F. Mancinelli and P. Inverardi, "Quantitative Resource-Oriented Analysis of Java (Adaptable) Applications," in *Proceedings of the 6th international workshop on Software and Performance*, Buenes Aires, Argentina, 2007.

[111] O. Marin, M. Bertier, P. Sens, Z. Guessoum, and J.-P. Briot1, "DARX - A Self-Healing Framework for Agents " *Lecture Notes in Computer Science,* June 21 2007.

[112] M. L. Massie, B. N. Chun, and D. E. Culler, "The Ganglia Distributed Monitoring System: Design, Implementation and Experience," *Parallel Computing,* vol. 30, pp. 817-840, July 2004.

[113] T. Masters, *Practical Neural Network Recipes in C++*. San Diego, CA, USA: Academic Press Professional, Inc., 1993.

[114] A. Messer, I. Greenberg, P. Bernadat, D. Milojicic, D. Chen, T. J. Giuli, and X. Gu, "Towards a Distributed Platform for Resource-Constrained Devices," in *Proceedings of the 22nd International Conference on Distributed Computing Systems*, Vienna, Austria, 2002, pp. 43-51.

[115] D. S. Milojicic, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou, "Process Migration," *ACM Computing Surveys,* vol. 32, pp. 241-299, 2000.

[116] R. Mohan, J. R. Smith, and C.-S. Li, "Adapting Multimedia Internet Content for Universal Access," *IEEE Transactions on Multimedia* vol. 1, pp. 104-114, March 1999.

[117] A. L. u. d. Moura, C. Ururahy, R. Cerqueira, and N. Rodriguez, "Dynamic Support for Distributed Auto-Adaptive Applications," 2002.

[118] A. Mukhija and M. Glinz, "Runtime Adaptation of Applications Through Dynamic Recomposition of Components," in *Proc. of 18th International Conference on Architecture of Computing Systems (ARCS 2005)*, 2005.

[119] C. Nester, M. Philippsen, and B. Haumacher, "A More Efficient RMI for Java," in *Proceedings of the ACM 1999 conference on Java Grande*, San Francisco, California, United States, 1999, pp. 152-159.

[120] B. Noble, "System Support for Mobile, Adaptive Applications," *IEEE Personal Communications,* vol. 7, pp. 44-49, Feb 2000.

[121] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker, "Agile Application-Aware Adaptation for Mobility," in *Proceedings of the 16th ACM Symposium on Operating Systems and Principles SOSP*, Saint-Malo, France, 1997, pp. 276-287.

[122] S. Numata, T. Itao, T. Ogawa, M. Tsukamoto, and S. Nishio, "Ja-Net on Grid for Adaptive Resource Distribution in Grid Environment," in *Proceedings of the 2005 International Conference on Active Media Technology*, Kagawa, Japan, 2005, pp. 409-414.

[123] S. Omar, X. Zhou, and T. Kunz, "Mobile Code, Adaptive Mobile Applications, and Network Architectures," in *Proceedings of the Second International Workshop on Mobile Agents for Telecommunication Applications (MATA 2000)*, Paris, France, 2000, pp. 17-28.

[124] OSGi Alliance, "OSGi," 2010. http://www.osgi.org/

[125] O. Othman, J. Balasubramanian, and D. C. Schmidt, "Performance Evaluation of an Adaptive Middleware Load Balancing and Monitoring Service," in *Proceedings of the 24th IEEE Intl. Conference on Distributed Computing Systems* Tokyo, Japan, 2004, pp. 135-146.

[126] O. Othman and D. C. Schmidt, "Optimizing Distributed System Performance via Adaptive Middleware Load Balancing," in *Proceedings of the Workshop on Optimization of Middleware and Distributed Systems*, Snowbird, Utah, 2001.

[127] S. Ou, K. Yang, and A. Liotta, "An Adaptive Multi-Constraint Partitioning Algorithm for Offloading in Pervasive Systems," in *Proceedings of the Fourth Annual IEEE International Conference on Pervasive Computing and Communications*, Pisa, Italy, 2006, pp. 116-125.

[128] S. Ou, K. Yang, and J. Zhang, "An Effective Offloading Middleware for Pervasive Services on Mobile Devices," *Pervasive and Mobile Computing,* vol. 3, pp. 362-385, August 2007.

[129] I. Park, D. Lee, and S. J. Hyun, "A Dynamic Context-Conflict Management Scheme for Group-Aware Ubiquitous Computing Environments," in *Proceedings of the 29th Annual International Computer Software and Applications Conference*, Edinburgh, Scotland, UK, 2005, pp. 359-364.

[130] G. D. Parrington, "Reliable Distributed Programming in C++: The Arjuna Approach," in *Second Usenix C++ Conference*, 1990, pp. 37-50.

[131] S. Parsa and O. Bushehrian, "On the Optimal Object-Oriented Program Re-modularization," in *Proceedings of the 7th international conference on Computational Science*, Beijing, China, 2007, pp. 599-602.

[132] S. Parsa and V. Khalilpoor, "Automatic Distribution of Sequential Code Using JavaSymphony Middleware," in *Proceedings of 32nd Conference on Current Trends in Theory and Practice of Computer Science*, Merin, Czech Republic, 2006, pp. 440-450.

[133] N. Paspallis and G. A. Papadopoulos, "An Approach for Developing Adaptive, Mobile Applications with Separation of Concerns," in *30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, 2006, pp. 299-306.

[134] M. Philippsen and M. Zenger, "JavaParty - Transparent Remote Objects in Java," *Concurrency: Practice and Experience,* vol. 9, pp. 1225-1242, 1997.

[135] A. Popovici, A. Frei, and G. Alonso, "A Proactive Middleware Platform for Mobile Computing," in *Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware*, Rio de Janeiro, Brazil 2003, pp. 455-473.

[136] P. Pratikakis, J. Spacco, and M. Hicks, "Transparent proxies for java futures," in *Conference on Object Oriented Programming Systems Languages and Applications*, Vancouver, BC, Canada 2004, pp. 206-223.

[137] P. Prekop and M. Burnett, "Activities, Context and Ubiquitous Computing," *Computer Communications,* vol. 26, pp. 1168-1176, July 1 2003.

[138] T. A. Proebsting and S. A. Watterson, "Krakatoa: Decompilation in Java," in *Proceedings of the 3rd Conference on USENIX Conference on Object-Oriented Technologies (COOTS)*, Portland, Oregon, 1997, p. 14.

[139] R. Quitadamo, G. Cabri, and L. Leonardi, "Mobile JikesRVM: A Framework to Support Transparent Java Thread Migration," *Science of Computer Programming,* vol. 70, pp. 221-240, February 2008.

[140] A. Ranganathan and R. H. Campbell, " An infrastructure for context-awareness based on first order logic," *Personal and Ubiquitous Computing,* vol. 7, pp. 353-364, December 2004.

[141] R. Reichle, M. Wagner, M. U. Khan, K. Geihs, M. Valla, C. Fra, N. Paspallis, and G. A. Papadopoulos, " A Context Query Language for Pervasive Computing Environments," in *Proceedings of the 2008 Sixth Annual IEEE International Conference on Pervasive Computing and Communications*, Hong Kong, 2008.

[142] J. S. Rellermeyer, G. Alonso, and T. Roscoe, "R-OSGi: Distributed Applications Through Software Modularization," in *Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware*, Newport Beach, California, 2007, pp. 1-20.

[143] M. Roman, F. Kon, and R. H. Campbell, "Reflective Middleware: From Your Desk to Your Hand," *IEEE Distributed Systems Online,* vol. 2, 2001.

[144] P. Rossi and C. Ryan, "An Empirical Evaluation of Dynamic Local Adaptation for Distributed Mobile Applications " in *International Symposium on Distributed Objects and Applications (DOA 2005)*, Agia Napa, Cyprus, 2005.

[145] H. Rubinsztejn, M. Endler, and N. Rodrigues, "A Framework for Building Customized Adaptation Proxies," in *Proc. of the IFIP conference on Intelligence in Communication Systems (INTELLCOMM 2005).* 2005.

[146] C. Ryan and P. Rossi, "Software, Performance and Resource Utilisation Metrics for Context-Aware Mobile Applications," in *Proceedings of International Software Metrics Symposium*, Como, Italy, 2005.

[147] C. Ryan and C. Westhorpe, "Application Adaptation through Transparent and Portable Object Mobility in Java," in *International Symposium on Distributed Objects and Applications (DOA 2004)*, Larnaca, Cyprus, 2004.

[148] K. Sakamoto and M. Yoshida, "Design and Evaluation of Large Scale Loosely Coupled Cluster-based Distributed Systems," in *IFIP International Conference on Network and Parallel Computing Workshops (NPC 2007)*, Dalian, China, 2007, pp. 572-577.

[149] F. Sanen, E. Truyen, W. Joosen, A. Jackson, A. Nedos, S. Clarke, N. Loughran, and A. Rashid, "Classifying and Documenting Aspect Interactions," in *Proceedings of the Fifth AOSD Workshop on Aspect, Components, and Patterns for Infrastructure Software*, Bonn, Germany, 2006, pp. 23-26.

[150] M. Satyanarayanan, "Mobile Information Access," *IEEE Personal Communications,* vol. 3, pp. 26-33, 1996.

[151] B. N. Schilit, N. Adams, and R. Want, "Context-Aware Computing Applications," in *Proceedings of the 1994 First Workshop on Mobile Computing Systems and Applications*, Santa Cruz, California, USA, 1994, pp. 85-90.

[152] B. N. Schilit and M. M. Theimer, "Disseminating Active Map Information to Mobile Hosts," *IEEE Network,* vol. 8, pp. 22-32, 1994.

[153] A. Schmidt, "Ontology-Based User Context Management: The Challenges of Imperfection and Time-Dependence," in *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE*, Montpellier, France, 2006, pp. 995-1011.

[154] A. Schmidt, M. Beigl, and H.-w. Gellersen, "There is More to Context than Location," *Computers and Graphics,* vol. 23, pp. 893-901, December 1999.

[155] M.-T. Segarra and F. André, "A Framework for Dynamic Adaptation in Wireless Environments," in *Proceedings of the Technology of Object-Oriented Languages and Systems*, St. Malo, France, 2000, p. 336.

[156] M. Shapiro, P. Dickman, and D. Plainfosse, "SSP Chains: Robust, Distributed References Supporting Acyclic Garbage Collection," in *ACM Symposium on Principles of Distributed Computing*, Vancouver, 1992.

[157] F. J. d. S. e. Silva, M. Endler, and F. Kon, "Developing Adaptive Distributed Applications: A Framework Overview and Experimental Results," in *Proceedings of the International Symposium on Distributed Objects and Applications CoopIS/DOA/ODBASE*, 2003, pp. 1275-1291.

[158] L. Skorin-Kapov and M. Matijasevic, "Dynamic QoS negotiation and Adaptation for Networked Virtual Reality Services," in *Sixth IEEE International Symposium on World of Wireless Mobile and Multimedia Networks (WoWMoM 2005)*, Taormina, Giardini Naxos, Italy, 2005, pp. 344-351.

[159] J. P. Sousa, V. Poladian, D. Garlan, B. Schmerl, and M. Shaw, "Task-based Adaptation for Ubiquitous Computing," *IEEE Transactions on Systems, Man, and Cybernetics,* vol. 36, pp. 328-340, May 08 2006.

[160] A. Spiegel, "Pangaea: An Automatic Distribution Front-End for Java," in *Proceedings of the 4th IEEE Workshop on High-Level Parallel Programming Models and Supportive Environment*, San Juan, Puero Rico, USA, 1999, pp. 93-99.

[161] C. Steketee, W. P. Zhu, and P. Moseley, "Implementation of Process Migration in Amoeba," in *Proceedings of the 14th International Conference of Distributed System*, Hsinchu, Taiwan, Republic of China, 1994, pp. 194-201.
[162] S. S. Stevens, "On the Theory of Scales of Measurement," *Science,* vol. 103, pp. 677-680, June 7 1946.
[163] M. Stoer and F. Wagner, "A Simple Min-Cut Algorithm," *Journal of the ACM,* vol. 44, pp. 585-591, July 1997.
[164] T. Strang and C. Linnhoff-Popien, "A Context Modeling Survey " in *First International Workshop on Advanced Context Modelling, Reasoning and Management*, Nottingham, England, 2004.
[165] T. Strauss and O. Theel, "Integration of a Dynamic Object Replication Framework in Java," in *Proceedings of the Eleventh Euromicro Conference on Parallel, Distributed and Network-Based Processing*, Genova, Italy, 2003, pp. 258-265.
[166] StringTemplate, "StringTemplate," 2010. http://www.stringtemplate.org/
[167] C. Sun, S. Xia, D. Sun, D. Chen, H. Shen, and W. Cai, "Transparent Adaptation of Single-User Applications for Multi-User Real-Time Collaboration," *ACM Transactions on Computer-Human Interaction (TOCHI),* vol. 13, pp. 531 - 582, 2006.
[168] Sun Developer Network, "Sun Bug Database," 2010. http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6902845
[169] Sun Microsystems, "Applets," 1994-2010. http://java.sun.com/applets/
[170] Sun Microsystems, "Dynamic Proxy Classes," 1994-2010. http://java.sun.com/j2se/1.3/docs/guide/reflection/proxy.html
[171] Sun Microsystems, "Enterprise JavaBeans Technology," 1994-2010. http://java.sun.com/products/ejb/
[172] Sun Microsystems, "Java EE," 1994-2010. http://java.sun.com/javaee/index.jsp
[173] Sun Microsystems, "Remote Method Invocation," 1994-2010. http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp
[174] T. Suzuki, T. Izumi, F. Ooshita, and T. Masuzawa, "Self-Adaptive Mobile Agent Population Control in Dynamic Networks Based on the Single Species Population Model," *IEICE Transactions on Information and Systems* vol. E90-D, pp. 314-324, January 2007 2007.
[175] S. Tang, J. Yang, and Z. Wu, "A Context Quality Model for Ubiquitous Applications," in *Proceedings of the 2007 IFIP International Conference on Network and Parallel Computing*, Dalian, China, 2007, pp. 282-287.
[176] E. Tanter, M. Vernaillen, and J. Piquer, "Towards Transparent Adaptation of Migration Policies," in *8th ECOOP Workshop on Mobile Object Systems (EWMOS 2002)*, 2002.
[177] M. Tatsubori, T. Sasaki, S. Chiba, and K. Itano, "A Bytecode Translator for Distributed Execution of ``Legacy'' Java Software," in *Proceedings of the 15th European Conference on Object-Oriented Programming*, Budapest, Hungary, 2001, pp. 236-255.
[178] E. Tilevich and Y. Smaragdakis, "J-Orchestra: Automatic Java Application Partitioning," in *Proceedings of the 16th European Conference on Object-Oriented Programming*, Malaga, Spain, 2002, pp. 178-204.
[179] E. Truyen, B. Robben, B. Vanhaute, T. Coninx, W. Joosen, and P. Verbaeten, "Portable Support for Transparent Thread Migration in Java," in *Proceedings of the Second International Symposium on Agent Systems and Applications*, Zurich, Switzerland, 2000, pp. 29-43.
[180] J. Vallejos, E. G. Boix, E. Bainomugisha, P. Costanza, W. D. Meuter, and E. Tanter, "Towards Resilient Partitioning of Pervasive Computing Services," in *Proceedings of the 3rd ACM workshop on Software Engineering for Pervasive Services*, Sorrento, Italy, 2008, pp. 15-20.
[181] R. Veldema, R. F. H. Hofman, R. A. F. Bhoedjang, C. J. H. Jacobs, and H. E. Bal, "Source-Level Global Optimizations for Fine-Grain Distributed Shared Memory Systems," in *Proceedings of*

the eighth ACM SIGPLAN symposium on Principles and Practices of Parallel Programming, Snowbird, Utah, United States 2001, pp. 83-92.

[182] E. Visser, "Meta-Programming with Concrete Object Syntax," in *Proceedings of the 1st ACM SIGPLAN/SIGSOFT conference on Generative Programming and Component Engineering*, Pittsburgh, PA, USA, 2002, pp. 299-315.

[183] E. Visser, "Stratego: A Language for Program Transformation Based on Rewriting Strategies," in *Proceedings of the 12th International Conference on Rewriting Techniques and Applications*, Utrecht, The Netherlands, 2001, pp. 357-362.

[184] M. A. Vouk, "Cloud computing — Issues, research and implementations," in *Information Technology Interfaces*, Dubrovnik, 2008, pp. 31-40.

[185] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall, *A Note on Distributed Computing* vol. 1222: Springer-Verlag London, UK 1996.

[186] L. Wang and M. Franz, "Automatic Partitioning of Object-Oriented Programs for Resource-Constrained Mobile Devices with Multiple Distribution Objectives," in *Proceedings of the 2008 14th IEEE International Conference on Parallel and Distributed Systems*, Melbourne, Victoria, Australia, 2008, pp. 369-376.

[187] X. H. Wang, D. Q. Zhang, T. Gu, and H. K. Pung, "Ontology Based Context Modeling and Reasoning using OWL," in *Proceedings of the Second IEEE Annual Conference on Pervasive Computing and Communications Workshops*, Orlando, Florida, 2004, p. 18.

[188] R. Want, A. Hopper, V. Falcão, and J. Gibbons, "The Active Badge Location System," *ACM Transactions on Information Systems,* vol. 10, pp. 91-102, January 1992.

[189] L. Wujuan and B. Veeravalli, "Design and Analysis of an Adaptive Object Replication Algorithm in Distributed Network Systems," *Computer Communications,* vol. 31, pp. 2005-2015, June 2008.

[190] L. Youseff, M. Butrico, and D. Da Silva, "Toward a Unified Ontology of Cloud Computing," in *Grid Computing Environments Workshop*, Austin, TX, US, 2008, pp. 1-10.

[191] K. Zhang and S. Pande, "Efficient Application Migration under Compiler Guidance," in *Language, Compiler and Tool Support for Embedded Systems*, Chicago, Illinois, USA, 2005, pp. 11 - 20.

[192] W. Zhu, C.-L. Wang, and F. C. M. Lau, "JESSICA2: A Distributed Java Virtual Machine with Transparent Thread Migration Support," in *Proceedings of 2002 IEEE International Conference on Cluster Computing*, Chicago, USA, 2002, pp. 381-388.