THE IMPACT OF RUNTIME METRICS COLLECTION ON ADAPTIVE MOBILE APPLICATIONS

HENDRIK GANI

Supervisor: DR. CASPAR RYAN

Honours Thesis

School of Computer Science and Information Technology
RMIT University
Melbourne, AUSTRALIA

October, 2005

Abstract

This thesis proposes, implements, and evaluates in terms of correctness and performance, an online metrics collection strategy to assist mobile application adaptation with the support of a mobile object framework. The solution is based upon an abstract representation of the system, which holds containers aggregating metrics for each specific component comprising a mobile object framework, including host mangers, runtimes and mobile objects. A key feature of the solution is the specification of multiple configurable criteria to control the propagation of metrics through the system, thereby reducing collection overhead. The MobJeX platform was used as the basis for implementation and synthetic applications were evaluated in order to provide an initial performance benchmark in terms of metrics collection overhead for a subset of a recently published suite of metrics. Preliminary findings demonstrate that even with simple collection criteria it should be possible to achieve positive adaptation outcomes that offset the overhead of the metrics collection process.

Contents

List of Figures

1 Introduction and Rationale

The heterogeneity and dynamic nature of the runtime environments in which applications execute, presents major challenges for the development and efficient execution of distributed mobile applications.

Different device capabilities in terms of processing power, storage capacity, display size and resolution, network connectivity, and battery power, limit the effectiveness of "write once run everywhere" applications, even those written in portable programming languages such as Java (Sun Microsystems, 2005) and .NET (Microsoft, 2005). This is because even if portability is achieved with minimal code change, the varying capabilities of the target devices and the networks in which they operate will result in different and most likely sub-optimal application behaviour.

Furthermore, in addition to this heterogeneity, the execution environment of mobile applications is extremely dynamic. For example, strict power consumption requirements as a battery becomes drained can result in reduced processing power, display brightness and connectivity. Additionally, the mobile networks in which typical mobile devices and their applications operate, are subject to varying speed, error rate and frequent disconnection as the devices move and network signal strength changes, or users connect and disconnect (Katz, 1994). As such, it is simply not possible to predict in advance the exact environment in which an application may run and thus there is much research to be done concerning the effective development and deployment of mobile applications. One particular solution which is receiving increasing attention in the research community is adaptive applications (Flinn and Satyanarayanan, 1999) (Gazit et al., 2000) (Hirschfeld and Kawamura, 2004) (Lemlouma and Layaïda, 2004) (Noble et al., 1997) (Ryan and Westhorpe, 2004) (da Silva e Silva et al., 2003)

In the general sense, application adaptation refers to the ability of an application to adjust its behaviour at runtime whenever the application senses changes in its environment, such as network bandwidth, processing load or free memory. Ideally, an adaptive application should be able to adapt to these changes without human intervention (Noble et al., 1997), especially if the application is running in a highly dynamic environment.

Application adaptation can be used to address both the issues of heterogeneity and the dynamic nature of the environment. Consider the example of an application designed to execute on a dedicated high speed LAN. If this application were to be run on a slower network such as 802.11 or an even slower 3G cellular network, one possible way to maintain the responsiveness of the application would be by lessening the data transferred over the network (i.e. adapting the application) at the expense of limiting the functionality of the application. Ideally, the application would be able to detect the change and adapt its behaviour accordingly without user intervention. Also consider that once the application has left the LAN and become mobile, it is subject to a much more dynamic environment and thus more frequent fluctuations in connectivity and network throughput, which can again be responded to via adaptation. The application could also respond to other dynamic events such as low battery power, in which case it could avoid performing operations involving high processing intensity or network transfer in order to conserve battery power (Chen et al., 2004) (Farkas et al., 2000) (Flinn and Satyanarayanan, 1999).

Application adaptation can be divided into two broad categories: service-based adaptation (Narayanan et al., 2000) (Noble et al., 1997) (Hirschfeld and Kawamura, 2004) and mobility based adaptation (Ryan and Westhorpe, 2004). Service-based adaptation is achieved by adjusting service quality or service behaviour in response to changes in the runtime environment of the application (Hirschfeld and Kawamura, 2004). In contrast, adaptation via mobility is achieved by moving software components such as objects to different hosts in such a way that application efficiency can be improved (in terms of performance versus resource utilization) when it is adversely impacted by environmental

changes.

In order for an adaptive application to be able to detect changes in the environment, the application needs to be able to collect resource information using a resource monitoring subsystem (da Silva e Silva et al., 2003). Additionally, information about application behaviour can also be useful for making adaptation decisions (Chen et al., 2004). This information, both about the environment and the software itself, needs to be represented accurately and thus each specific attribute needs to be formally defined and then quantified using explicit metrics. Metrics related to resource information are referred to as *environmental* metrics (Ryan and Rossi, 2005), while metrics representing application behaviour are referred as *software* metrics (Rossi and Fernandez, 2003). Metrics are essential for application adaptation, because they accurately represent the information needed for performing adaptation, and thus the runtime collection of metrics, especially those related to software, is the main focus of this thesis.

Note however that the type of metrics required for performing adaptation is dependent upon the type of adaptation required. For example, service-based adaptation may require detailed *environmental* metrics but only *simple software* metrics. As an example, consider a case of Quality of Service (QoS) adaptation (a specific type of service-based adaptation) in which a simple Video-on-Demand (VoD) server decides the rate of video frame delivery based on network link speed and the specified video quality requirement in terms of frame resolution (Noble et al., 1997). On the other hand, for more sophisticated adaptation, such as adaptation via mobility, *detailed* software metrics (Ryan and Rossi, 2005) such as those described in section 2 and appendix A are needed, given the more complex computations involved in the adaptation decision making process. This is because the placement of specific objects is dependent not only on the metrics of the given environment (network and host capabilities in terms of throughput, storage, processing power, memory availability etc.), but on specific characteristics of the software components to be moved. Examples of these software characteristics include the processing intensity of methods and the communication overhead between objects as discussed in section 3.

Although metrics are essential for adaptive application decision making, their collection process imposes different overheads depending on the collection strategy. These overheads include polling the system periodically to capture its state (Narayanan et al., 2000) (Rolia and Lin, 1994), sending event notification when a change is detected (da Silva e Silva et al., 2003) or deriving a metric from combination of other metrics (Ryan and Rossi, 2005). Therefore, it is important to choose efficient strategies for collecting these metrics, which is the main focus of this research.

Because contemporary mobile applications are predominately object-oriented, mobile object frameworks (Ryan and Westhorpe, 2004) (Voyager, 2005) (Holder et al., 1999) (Philippsen and Zenger, 1997) can be used to *assist* the implementation of application adaptation through mobility. Objects of a non-mobile application can be made mobile by porting them to run in a mobile object framework. Existing mobile object frameworks include MobJeX (Ryan and Westhorpe, 2004), Voyager (Voyager, 2005), FarGo (Holder et al., 1999), and JavaParty (Philippsen and Zenger, 1997) as reviewed by Ryan and Westhorpe (Ryan and Westhorpe, 2004). However, none of these frameworks, with the exception of MobJeX, provide an adaptation engine nor the metrics collection process required to support such adaptation. Consequently, mobility or adaptation operations must be pre-determined either via scripting or explicit code within the application itself. In the case of MobJeX, a prototype of a simple local adaptation engine (Rossi and Ryan, 2005) has been implemented, however this was tested using synthetic metrics because there is little prior work on the dynamic collection of software metrics, and no existing means of automatically collecting them. This is discussed further in the literature review in section 2 and as a result, proposing a solution for dynamic metrics collection, and implementing and testing it in the MobJeX framework, serves as the main contribution of this thesis. This problem

is non-trivial since mobile object frameworks consist of multiple application components, and thus metrics collection must be performed and communicated amongst these components. Furthermore, in some cases the location where each metric should be collected is not fixed (i.e. it could be done in several places) and thus a decision must be made based on the efficiency of the chosen solution. This is described in detail in section 3.2.

Another important aspect of the metrics collection process is the delivery or propagation of metrics to the adaptation engine. Due to the variations in the locations where metrics are collected, there has to be a mechanism to pass these metrics to an adaptation engine to facilitate the decision making process. Furthermore, since the communication between these components and the adaptation engine impacts on application performance, an efficient strategy for presenting the metrics to the adaptation engine is needed. Consequently, such a strategy is proposed in section 3.6. The most effective way to test the metrics collection strategies presented in this thesis is to implement them and integrate them into an existing adaptation framework. The MobJeX framework (Ryan and Westhorpe, 2004), which is described further in section 2.2.1.1, was chosen for this purpose since it is the most recent and active framework for the support of adaptation via object mobility and thus its adaptation requires sophisticated metrics collection within the objects of an application, the techniques nor the implementation for which currently exist. The MobJeX framework does have a resource monitoring facility (da Silva e Silva et al., 2003), which simplifies the testing of the collection of environmental, although it does not provide a means for propagating this information to the adaptation engine. As such, an efficient solution to this problem is also provided in section 3.6.

Having provided a rationale for the adaptation of distributed mobile applications and having discussed the challenges of collecting the environmental and software metrics required for such adaptation, this thesis aims to answer the following research questions:

1. How can such metrics be collected?

   The collection process involves both the measurement itself and the propagation of these metrics to the adaptation engine. This question was addressed by proposing collection strategies for each of the important metrics in section 3, implementing them, and testing their correctness in section 4.

2. What is the overhead of the metrics collection process?

   The overhead of the metrics collection process developed in response to question 1 was identified by code instrumentation using a number of different scenarios involving: varying number of objects; number of methods per object, and method behaviour in terms of processing intensity and parameter size. These results are described in section 4.

3. How can the overhead of the metrics collection process be reduced?

   This was addressed by providing flexible criteria as part of the solution addressing research question 1, in order to reduce collection overhead by only delivering metrics that have changed substantially or are likely to have an impact on the adaptation process. These criteria are discussed in 3.7 with some initial implementations tested as part of the experimental evaluation of section 4.

CONTRIBUTION

In answering the above research questions, the main contribution of this thesis has been to propose, test and evaluate a solution for collecting both environmental and software metrics and delivering them to an adaptation engine. Furthermore, by implementing these solutions and integrating them into the

MobJeX framework (Ryan and Westhorpe, 2004), this thesis directly facilitates further research on the increasingly important area of mobile application adaptation. In addition, the design of a metrics collection and propagation strategy based on variable criteria encourages further work looking at the specification of more detailed criteria based on the temporal characteristics of metrics, in order to further reduce the impact of the metrics collection process and thus continue to increase the benefits in performance that can be achieved via adaptation.

The rest of this thesis is organised as follows: section 2 contains background information about the importance of metrics and runtime metrics collection, while section 3 describes the derivation of solutions to the problem of collecting such metrics. Section 4 is concerned with testing the proposed solution by providing experimental results and analyses, and finally, section 5 presents conclusions and discusses possible future work in the area of metrics collection and application adaptation.

2 Literature Review

This section provides a detailed literature review of the two major aspects of the metrics collection process. The first aspect, the actual measurement (or derivation or estimation) of the metric is discussed in subsection 2.1, whilst the second part, that is the propagation and delivery of metrics to the adaptation engine, is covered in subsection 2.2.

2.1 Measurement

In order to provide a context for the review of existing work on metrics collection, and to facilitate further comparison with the research presented in this thesis, this section is presented in two parts. Section 2.1.1 provides background information concerning a number of important concepts related to the collection of metrics: namely metrics and granularity (2.1.1.1); static versus dynamic metrics (2.1.1.2); online versus offline collection (2.1.1.3); and estimation versus direct measurement (2.1.1.4). Following this background discussion section 2.1.2 presents the literature review proper as a comparative analysis of existing work on metrics collection with an emphasis on how this thesis extends and improves upon this existing body of work. In order to provide a scope relevant to this thesis, only the collection of software and environmental metrics are covered.

2.1.1 Background

2.1.1.1 Metrics and Granularity

The term measurement *attribute* refers to a specific characteristic of a system that is to be measured, whereas the term *metric* refers to an explicit measurement that quantifies the attribute in terms of units such as time, size, or capacity. Metrics can be defined to measure virtually any property however as discussed in the introduction, the metrics of interest in this thesis are those related to software and its execution environment, in order to facilitate application adaptation.

Of most relevance to the work in this thesis are the metrics of Ryan and Rossi (Ryan and Rossi, 2005), which were derived explicitly for the purpose of mobile application adaptation. Note however, that their paper addressed the derivation and evaluation of the metrics and how they could be used in adaptation, rather that how the metrics can actually be collected efficiently in practice as is the focus of this thesis. Nevertheless, their metrics form the basis of the solution presented in section 3 because they are the most recent and most comprehensive suite of metrics designed for adaptation; have been derived from a series of mathematical models and empirically validated, and have been empirically

shown to improve the application performance in an adaptation scenario compared with a random non-adapted base case.

In terms of metrics for the adaptation of mobile applications, the granularity of the measurement attributes and their corresponding metrics is important. Consider the case of measuring memory consumption. At a coarse level of granularity this could be measured for an entire application or even a system, but could also be measured at the level of an individual object; or for an even finer level of granularity, the memory consumption during the execution of a specific method could be measured. As an example of the level of granularity required for mobility based adaptation, the global adaptation algorithm proposed by Ryan and Rossi (Ryan and Rossi, 2005) requires metrics representing both the duration of a method execution and the overhead of a method invocation. Therefore, section 3.5 of this thesis proposes strategies to collect metrics at levels of granularity ranging from a single machine down to the individual method level.

2.1.1.2 Static versus Dynamic Metrics

Dynamic metrics are those metrics which change during the execution of an application which means that they need to be measured repeatedly, ideally only when their values change or are likely to have an impact on the adaptation decision making process. Conversely, static metrics do not change during the execution of the application (and in some cases may not change until the application is recompiled) and as such need only be collected once. Given this distinction, different strategies would generally be used for collecting static and dynamic metrics, as described in the follow subsection.

An example of a static metric from Appendix A is Executable Code Size which need only be collected once after an application is compiled and the class sizes are known. Alternatively, Network Usage can be expected to fluctuate throughout the execution of an application and thus this metric is dynamic, meaning it should be measured throughout the application lifetime.

2.1.1.3 Online versus Offline Collection

This thesis refers to metrics collection performed before an application runs as *offline* metrics collection, and metrics collection at runtime as *online* metrics collection.

As mentioned in the previous section, static metrics do not change during application execution and in most cases can be collected offline. However because dynamic metrics can change at runtime they must by necessity be measured online, the most significant impact being that in contrast to offline collection, online collection imposes an overhead in terms of application execution. As a result, this thesis is concerned with providing a solution for efficient online metrics collection, and thus only dynamic metrics are considered in the solution (section 3) and evaluation sections (section 4) of this thesis.

2.1.1.4 Estimation versus Direct Measurement

As stated in the previous section, obtaining dynamic metrics accurately must be done using *direct measurement* and online collection. However it is possible to perform an offline alternative in the form of metric *estimation*. For example, establishing the object coupling measurement of an application could be done offline by analysing its source code (Chen et al., 2004), and although this is less accurate than direct online measurement at runtime (since it fails to consider factors like polymorphism and unused or dead code) it is also more efficient because it does not impose an overhead on the running application.

Therefore on one hand, an accurate metric is preferable in order to support better application adaptation. However, on the other hand, the cost of collecting the metric should not be so high that in the end, the metric does not improve the overall performance or resource utilisation of the system. Therefore, ideally the best metrics collection strategy would be determined based on its effectiveness on the adaptation outcome, which is dependent on the trade-off between the accuracy and overhead of a given strategy. Note this thesis uses only the direct measurement of metrics since it is generally expected to be more accurate, leaving a comparison with metric estimation to future work.

2.1.2 Previous Work

There are many existing studies involving the *offline* collection of metrics, however they are primarily concerned with structural attributes such as size, complexity, and coupling (Morasca, 1999), or external attributes such as quality (Rossi and Fernandez, 2003). However, none of these studies consider offline measurement of dynamic performance metrics through estimation, and are therefore not relevant to a study of metrics collection for the adaptation of mobile applications, as is the objective of this thesis.

In terms of *online* metrics collection, there have been a number of earlier papers proposing simple adaptation based on environmental metrics collected at the machine or process level of granularity. For example in (Noble et al., 1997), a technique was proposed to support application aware adaptation (Jing et al., 1999) by measuring network bandwidth, CPU load, and battery power and delivering it to applications so that they could make adaptation decisions. Note that since the adaptation was application aware, namely carried out by the application with no framework or middleware support, there was by definition no automated collection of software metrics as is the goal of this thesis. In a different case, Flinn and Satyanarayanan (Flinn and Satyanarayanan, 1999) proposed the use of an environmental metric battery power, in order to assist energy-aware adaptation based on a trade-off between energy conservation and application quality for mobile applications. However, since both these papers were concerned only with simple machine or process level metrics, neither of the studies provide details about the metric collection process and as such are not highly influential upon the work described in this thesis.

A study by McGregor et al. (McGregor et al., 1999) is one of the earlier papers to consider the online collection of dynamic *software* metrics, in terms of number of invocations and response time for methods. However, although the study attempts to measure a method level metric, it is done at the process level outside of the object implementing the method. As such, it can only measure the response time of a single method call accurately if the call is only one level deep and as such is not suitable for the adaptation of applications with mobile components. A further limitation of this strategy is that it cannot be used for collecting metrics such as execution time (ET) and invocation time (IT) that require modification of the original method in order to insert measurement code (see section 3.2).

Although the paper describing the proposed adaptation engine for FarGo (Gazit et al., 2000), considers online collection of both software and environmental metrics, it provides neither a working implementation nor discusses how such metrics could be collected. Additionally, the software metrics proposed by the paper are collected only at the *complet* (a component cluster) level (Gazit et al., 2000) compared with the more fine grained metrics collection strategies discussed in this thesis in section 3.5. However, the paper is unique in that it proposes a metrics initiation technique that is independent from the component collecting the actual metrics. In this specific case, the scenario involved the application requesting metrics from the resource monitor as required based on application specific needs. This thesis proposes a more general solution for the case of *application initiated collection* as

described in section 3.3.

Arisholm et al. (Arisholm et al., 2004) proposed a system for dynamically measuring object coupling, for the purpose of determining the structural properties of software from a software engineering standpoint, by collecting metrics via internal measurement within the object itself. However the chosen metrics, namely number of invocations and lines of code, are not on their own sufficient for performing the adaptation of mobile applications, with additional software, performance, and resource utilisation metrics being needed, hence the solutions described in this thesis. Nevertheless, the use of those metrics to define a coupling measure could potentially be useful for the adaptation of mobile applications; however this is a subject for future work.

Chen et al. (Chen et al., 2004) investigated object level metrics needed by adaptive applications to offload code compilation to different hosts by measuring compilation cost and class migration time. In addition, the software metric number of invocations was measured, as was the resource attribute of communication channel condition. Note however that the focus of the paper was primarily on the specific adaptation scenario of distributed compilation, and that the metrics collection process was simulated.

Silva et al. (da Silva e Silva et al., 2003) describe a framework for developing adaptive distributed applications including a discussion of the collection of both software and resource metrics. A number of aspects were raised that have been incorporated and extended in this thesis. Firstly, Silva et al. introduced the concept of monitoring parameters to specify the range of acceptable metrics, which is similar to the concept of thresholds proposed in section 3.7. Secondly, they introduced the idea of *interceptor* components that could sit between two distributed objects and collect response time information about individual methods. However, although this was a novel idea, the solution presented in section 3 provides a more generalised solution to collecting (and propagating) different metrics within the proxy and implementation components of distributed (and mobile) components.

In summary this subsection has reviewed existing studies on metrics collection, particularly in the case of application adaptation of distributed and mobile applications. In doing so it has identified a number of ideas that have been incorporated into the solution in section 3. However, for the most part this review has shown that a flexible strategy for measuring and delivering metrics is not described in existing literature.

2.2 Metrics Propagation and Delivery

Having discussed the actual measurement of metrics in the previous section, the present section is concerned with the process of delivering those metrics to an external processing entity such as an adaptation engine. As with the previous section, this subsection is structured in two parts with section 2.2.1 providing background information and section 2.2.2 presenting the comparative literature review itself.

2.2.1 Background

2.2.1.1 Mobile Object Frameworks

This subsection describes the general structure and implementation of mobile object frameworks in order to understand the challenges related to the propagation of metrics through the components of such a framework, and the eventual delivery of those metrics to an external processing entity such as an adaptation engine. In doing so this section provides background information for both the discussion of previous work in section 2.2.2 and the solution to existing challenges in section 3.

In general, an object-oriented application consists of objects collaborating to provide the functionality required by a given problem domain. Mobile object frameworks allow some of these objects to be tagged as mobile objects, providing support for such objects to be moved at runtime to other machines or hosts. There are two common applications of mobile object technology. Firstly, distributing application objects to increase performance and balance load, which would ideally be done automatically using an adaptation engine as described in the introduction. Secondly, moving objects (especially head objects such as a user interface) to follow a user as he or she moves from one device to another.

At a minimum, a mobile object framework with at least one running mobile application consists of the following components: runtimes, mobile objects, and proxies (Shapiro, 1986), although the terminology used by individual frameworks can differ (Ryan and Westhorpe, 2004) (Voyager, 2005) (Holder et al., 1999) (Philippsen and Zenger, 1997). An example of the relationship between these components can be seen in figure 1, which is a schematic diagram of the MobJeX framework (Ryan and Westhorpe, 2004) in which the *mobility subsystem* is an example of a mobile object framework. These components, and their relationships are described in general terms below, with MobJeX specific issues explicitly identified in the context of the discussion.

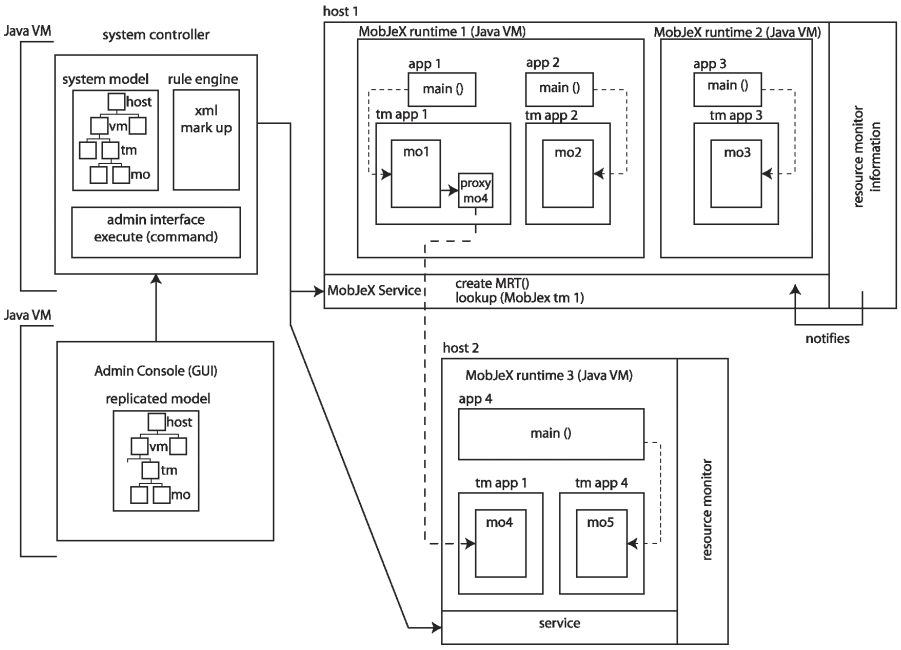

Figure 1: Architecture of the MobJeX framework (reproduced from Ryan and Westhorpe 2004 (Ryan and Westhorpe, 2004))

A *runtime* essentially corresponds to a process which can contain and manage mobile objects. In FarGo (Gazit et al., 2000) this component is know as a *core* and in most systems separate runtimes are required to allow different applications to run independently, although this is not the case with MobJeX, which can run multiple applications in a single runtime using threads.

In a mobile object application, objects interact with each other through proxies (Shapiro, 1986), which add remote communication and object tracking functionality to an object. Ideally, the proxy would be generated automatically without having to modify the original non-mobile object, as is the case with the MobJeX framework which, of the existing mobile object frameworks, provides

the highest level of transparency to the developer. In this thesis, we refer to the object making a method call as a *source object*, and the object whose method was invoked as the *target object*. To communicate with a target object, the source object needs to acquire a local proxy of the target object which is usually done when the target object is constructed. An additional proxy is required for each target object that it communicates with, however in the case of FarGo there is a single proxy (*tracker*) for each class rather than object.

The proxy of a mobile object has the same method interface as the object itself, and thus a source object can communicate with a target object without being aware that there is a proxy in between, thus simplifying the development process and the decision of whether an object should be mobile or not. The proxies themselves are also mobile, allowing them to be moved together with the source object when it moves to another host.

In a mobile object environment, a *host manager* can provide a central point of communication by running on a known port on a per host basis, in order to provide common functionality such as naming and directory services for enumerating or looking up components such as runtimes or mobile objects. Additionally, as described in section 3 and 4, the service can also reduce the overhead of propagating and delivering metrics from the mobile framework and application components to the adaptation engine. Of all the known mobile object frameworks, MobJeX (Ryan and Westhorpe, 2004) is the only one that provides a host manager component, known in this case as a MobJeX *service*. Nevertheless, the other frameworks (Voyager, 2005) (Holder et al., 1999) (Philippsen and Zenger, 1997), can provide some of the functionality of a host manager, such as naming and directory services, through runtime components. This however increases remote communication overhead and requires the use of pre-arranged ports between peer machines in the mobile environment. Therefore, to provide the most flexible solution to metrics collection, and since the solutions presented in this thesis are implemented and tested on MobJeX, a logical distinction is made between these two entities and thus in cases where a separate host manager does not exist, the runtime is assumed to take on this role.

There is one more aspect that distinguishes MobJeX from the general mobile object framework model of a single host manager co-ordinating one or more runtimes (depending upon on the number of running applications), which contain the mobile objects and their proxies forming the mobile application itself. That is, MobJeX has an additional container called a *transport manager* (TM) which provides a per-application based container within each runtime where there exists one TM for each application that has objects in a given runtime. Again for the reasons outlined above, the transport manager is considered as a separate component in this thesis, although this level of containment can be ignored, and the functionality ascribed to it handled by the runtime, in cases where a TM is not present.

2.2.1.2 Propagation and Delivery

As mentioned previously, mobile object frameworks including MobJeX consist of different application components (service, runtime, proxy, mobile object etc.) and each component has metrics of interest that need to be measured, and then presented periodically to the adaptation engine for processing. This thesis identifies two stages, the propagation of metrics through the components of the mobility framework and the delivery of those metrics from the host manager (or runtime if the host manager is not present) to the adaptation engine.

Note that this thesis identified two different approaches to gather metrics within the mobility framework itself, namely a push approach and a pull approach. In the pull approach, the adaptation engine or the host manager must poll or query each of the system components for their metrics. On the other hand, the push approach is an event notification based model (Gazit et al., 2000) (da Silva e

Silva et al., 2003), whereby under certain conditions interested parties are notified about changes in particular metrics. Given that this thesis is concerned with presenting an efficient model for the measurement, propagation and delivery of metrics; the push approach is chosen based on its advantages which are described in more detail in section 3.6. In brief, it is proposed that when a lower level system component detects the arrival of a new metric update (e.g. mobile object), the metric is pushed (possibly along with other relevant metrics) to the next level component (i.e. runtime or transport manager containing the mobile object), which at some later stage, again determined by a configurable criteria (for example when there are a sufficient number of changes) will get pushed to the next level component (i.e. the host manager or the adaptation engine).

Furthermore, propagation is treated separately from delivery since Ryan and Rossi (Ryan and Rossi, 2005) make the distinction between *local adaptation* and *global adaptation*, both of which are supported by metrics from the mobile object framework. Local adaptation is performed by an engine running on the local host (for example in MobJeX this would occur within the service) and thus in this case the delivery phase would be a local inter-process call. Conversely, global adaptation is handled by a centralised adaptation engine running on a remote host and thus the delivery of metrics is via a remote call, and in the case where multiple runtimes exist without a separate host manager the delivery process would be even more expensive. Therefore, due to the presence of network communication latency, it is important for the host manager to pass as many metrics as possible to the adaptation engine in one invocation, implying the need to gather these metrics in the host manager, through some form of push or propagation, before sending them to the adaptation engine.

2.2.1.3 Temporal Behaviour and History

Since, as described in the previous section, there are overheads associated with the propagation and especially the delivery of metrics, it is clear that they cannot be delivered too often. Rolia and Lin (Rolia and Lin, 1994) discuss the idea of a *process period* whereby metrics are stored (their system was simple and had no propagation) and delivered in a fixed interval, with the separation of storage (propagation) from delivery reducing the number of metrics delivered to the system without seriously affecting the *liveness* of the metrics. In this case, liveness refers to how much the real state of the system may have changed since the metrics were collected, with this property being a function of how old the metrics are and how rapidly the system is changing (i.e. its *temporal behaviour*). This thesis argues that process period alone is not the most effective means of controlling propagation and delivery whilst minimising the impact on metric liveness, and thus proposes in section 3.7 a technique for specifying flexible criteria to control the propagation and delivery of metrics.

Furthermore, since the adaptation process itself also incurs an overhead, adaptation should also not be performed too frequently. Ideally there should be a match between when metrics are sent to the adaptation engine and the likelihood that the adaptation engine will make an adaptation decision based on those metrics. For example, it is unlikely that adaptation should be performed in response to transient changes in the environment. Therefore this thesis discusses the maintenance of metrics history (Rolia and Lin, 1994) from which can be derived the behaviour of applications over time (i.e. its *temporal behaviour*) in order to aid in the efficient propagation and delivery of only those metrics that are likely to trigger adaptation. This process would also be controlled by the criteria introduced above.

2.2.2 Previous Work

Compared with the previous work on metrics collection and adaptation discussed in section 2.1.2, there is little existing literature concerning the delivery and propagation of metrics in complex environments such as mobile object frameworks. Furthermore, there exists little literature related to the storage of metrics history and the derivation of temporal behaviour from such history.

Nevertheless, although the idea of propagation and delivery is not present in the early work of Rolia and Lin (Rolia and Lin, 1994), they propose a technique which separates the measurement, storage and delivery of metrics to an external entity (the Distributed Application Performance Measurement System (DAPM)). The technique itself was simple involving the continuous accumulation of metrics at each node, with the metrics presented as a collection to the DAPM at pre-specified time intervals called a process period.

The paper by da Silva e Silva et al. (da Silva e Silva et al., 2003) is the only work reviewed which explicitly specifies a propagation model including monitoring packages, an event detection and notification package, and a dynamic reconfiguration package (adaptation engine). In their system, metrics were collected from a number of different sources and pushed immediately to the event detection and notification package, before being delivered at some future time to the adaptation engine based on specific criteria for each application. Although this is a novel approach, it has the disadvantage of having only one point (the event detection package) where a propagation decision can be made and thus there is still a lot of potentially unnecessary pushing from the various monitoring packages. This issue is addressed by the configurable criteria model presented in section 3.7 of this thesis.

Finally, although the study of Narayanan et al. (Narayanan et al., 2000) is unique in its proposed use of history to improve mobile application adaptation by analysing the past behaviour of an application, there was no discussion of how the history was recorded or how this could be done in an efficient way because the metrics collection was done offline and thus efficiency was not a concern of the paper. Metrics that were used in this study include resource utilisation metrics, such as CPU load and fidelity metrics that were specific to the media content used in their research. In contrast the aim of this thesis is to propose more general solutions and thus section 3 presents strategies for storing and efficiently querying metrics history in order to determine the temporal behaviour of application.

3 Solutions

The previous section identified the need for the efficient collection of metrics to support the adaptation of mobile applications, and noted that there is currently limited work in this area and thus many outstanding issues. Consequently, this section proposes and discusses the derivation of such a solution, with emphasis on why particular choices were made. Note that since this thesis is concerned primarily with the adaptation of mobile applications, the rest of this section considers *adaptation* to refer to the adaptation for mobile applications unless specified otherwise. Furthermore, the solution presented in this section was implemented on the MobJeX framework (Ryan and Westhorpe, 2004), as described in the literature review of section 2. Nevertheless, where possible these solutions are discussed in generic terms for any mobile object framework, except where explicitly stated to be MobJeX specific.

3.1 Metrics Selection

This subsection discusses some concrete metrics that were implemented and tested in this research to help evaluate the correctness and performance of the solution proposed in this thesis. As discussed in the literature review the metrics of Ryan and Rossi (Ryan and Rossi, 2005) have been chosen as the

basis for this solution (see Appendix A), since they are specifically intended for mobile application adaptation as well as having been derived from a series of mathematical models and empirically validated; and empirically shown to improve the application performance in a real adaptation scenario compared with a random non-adapted base case.

It would however be beyond the scope of this thesis to implement and test the full suite of metrics listed in Appendix A, and thus a smaller subset was chosen. In order to provide a useful non-random subset, we chose to implement, and test in section 4, the minimum set of metrics necessary to implement *global adaptation* (Ryan and Rossi, 2005) and thus satisfy an actual adaptation scenario. As such the solution presented in this section is discussed primarily in terms of these metrics, although the structure of the solution is believed to support the implementation of the remaining metrics, as well as other unspecified metrics such as those related to quality and resource utilisation. This subset, which is an extract from Appendix A is listed below and categorised according to metric type. Note that some additional metrics were used for implementation purposes in order to derive core metrics or assist the evaluation, and as such are defined in context where appropriate.

1. Software metrics
   - Number of Invocations (NI)
     The frequency of invocations on methods of a class.
2. Performance metrics
   - Method Execution Time (ET)
     The time purely taken to execute a method (in milliseconds)
   - Method Invocation Time (IT)
     The time taken to invoke a method, excluding the method execution time (in milliseconds).
3. Resource utilization metrics
   - Memory Usage (MU)
     The memory usage of the process an application is running on (in bytes).
   - Processor Usage (PU)
     The percentage of the CPU load of a host.
   - Network Usage (NU)
     The network bandwidth between two hosts (in bytes/sec).

The following paragraphs give a brief description of a number of these metrics in order to demonstrate their importance in an adaptation scenario.

As Processor Usage (PU) on a certain host increases, the Execution Time (ET) of a given method executed on that host also increases (Ryan and Rossi, 2005), and thus this measure is used to help the adaptation engine decide whether to move an object with high ET to another host with low PU.

Invocation Time (IT) shows the overhead of invoking a certain method. The invocation overhead of marshalling parameters and transmitting remote data for a remote call is order of magnitudes higher than the cost of pushing and popping data from the function call stack. In other words, remote method invocation is expensive and thus should be avoided unless the gains made by moving an object to a host with more processing power (thereby reducing ET) outweigh the higher IT of the remote call.

Number of Invocations (NI) is used primarily as a weighting factor or multiplier in order to enable the adaptation engine to predict the value over time of a particular adaptation decision.

3.2 Metrics Measurement

In order to answer Research Question 1, this section discusses how each of the metrics in the subset under investigation can be collected in terms of direct measurement or derivation, and where in the mobile object framework such metrics should actually be measured.

Of the environmental resource metrics, Processor Usage (PU) and Network Usage (NU) both relate to an individual machine, and thus can be directly measured through the resource monitoring subsystem that is instantiated as part of the MobJeX *service*. However, Memory Usage (MU), which represents the memory state of a running process rather than the memory usage of a host, should instead be collected within an individual *runtime*.

The measurement of Number of Invocations (NI) and Execution Time (ET) metrics can be also be performed via direct measurement, however in this case within the object implementation itself. NI involves simply incrementing a counter value at either the start or end of a method call, depending upon the desired semantics with regard to thrown exceptions, while ET can be measured by starting a timer at the beginning of the method and stopping it at the end of the method, then retrieving the duration recorded by the timer.

In contrast, collecting Invocation Time (IT) is not as straight forward because the time taken to invoke a method can only be measured after the method finishes its execution and returns to the caller. In order to collect IT metrics, another additional metric is needed. Ryan and Rossi (Ryan and Rossi, 2005) define the metric Response Time (RT), as the total time taken for a method call to finish, which is the sum of IT and ET. The Response Time can be measured directly using the same timer based technique used to measure ET, although at the start and end of the proxy call rather than the method implementation. Once the Response Time (RT) is known, IT can derived by subtracting RT from ET.

Although this derivation appears simple, in practice it is complicated by the fact that the RT and ET values from which the IT is derived are by necessity measured using timer code in different locations i.e. RT measured in the proxy, ET measured in the method body of the object implementation. In addition, the proxies are by definition not part of the MobJeX containment hierarchy, since although proxies have a reference to their target object, it is not efficient for a mobile object (mobject) to have backward references to all of the proxies which reference it. Fortunately, this problem can be solved using the push based propagation mechanism described in section 3.6 in which the RT metric is pushed to the mobject so that IT can be derived from the ET value stored there. The derived value of IT is then stored and propagated further as necessary according to the criteria of section 3.7.

Finally, note that the measurement of metrics, whether consisting of making a system timer call or calculating a derived metric, imposes an overhead and thus it should be considered within the scope of research question 2. Nevertheless, this overhead is generally small compared to that of propagating metrics through the mobility framework or delivering them to the adaptation engine as described in section 3.6.

3.3 Metrics Collection Initiation

As mentioned previously, metrics measurement can be initiated either by scheduled polling or as part of normal application flow. The decision of which initiation type to use depends on the types of metric to be are measured.

This thesis identifies the polling approach as most appropriate for collecting resource utilisation metrics, such as Processor Usage (PU), Network Usage (NU) and Memory Usage (MU), since they are not part of, or related to, the direct flow of the application. To measure PU or NU, the resource monitor polls the Operating System for the current CPU or network load respectively. In the case of

Memory Usage (MU), the Java Virtual Machine (JVM) (Lindholm and Yellin, 1996) is polled for the current memory load. Note that in order to minimise the impact on application response the polling action should be done asynchronously in a separate thread.

Metrics that are suitable for application initiated collection are software and performance related metrics, such as Number of Invocations (NI), Execution Time (ET), and Invocation Time (IT), which are explicitly related to the normal invocation of a method, and thus can be measured directly measured at this time.

3.4 Metrics Representation

In the implementation proposed in this thesis, different classes were used to store metrics, which include `NumericMetric`, `DecimalMetric`, and `DurationMetric`. Each of these specialised classes provides different methods related to the measurement and manipulation of metrics, which are specific to the subtype of metric in question. For example, a `DurationMetric` could have `start()`, `stop()`, `reset()` and `getDuration()` methods. Furthermore, all of the metric types implement a `Metric` interface, which defines general methods to access the stored metrics, allowing the retrieval of metrics from different `Metric` objects without the need to know the actual object type (the specific metric class). This approach not only allows access through a generic interface, but also allows the client to identify the type of metric an object represents by further extending the class for each specific type of metric, such as `NIMetric`, `ETMetric`, `MUMetric`, etc.

The `Metric` object also acts as a container for values that are collected over time, thus a history of the collected metrics can be maintained in the object. By maintaining the history of metric values, the `Metric` object can support more advanced querying functionality than simply requesting raw metric values. Examples of more sophisticated functionality include the number, mean, and standard deviation of the measured values. Furthermore, although not currently implemented, the object could also support queries for values that were collected over a specified time period or ranged between specified lower and upper limits, etc.

The other benefit of storing history inside the `Metric` objects is that it facilitates the derivation of the temporal behaviour. For example, a mathematical function representing the historically measured metric values of a particular resource could be presented to the adaptation engine in order to enable predictions of future resource consumption (Narayanan et al., 2000). However, determining the temporal behaviour of metrics is beyond the scope of this research and thus left to future work.

Care should be taken regarding history growth since some applications run for weeks, months, or even years, and thus the number of values added to the object can become very large, which not only impacts memory consumption but also increases query processing time. Possible solutions include flushing historical data to persistent storage (da Silva e Silva et al., 2003), or using a sliding window mechanism to discard old metrics as new metrics are added to the history collection. The argument in favour of this latter approach is that older metrics are less representative of the system and application states and thus less relevant.

3.5 Metrics Aggregation

This thesis proposes a solution where all metrics collected in the same location are aggregated in a metrics container with individual containers corresponding to functional components in the mobile object framework. The following is a list of metrics containers and the metrics that are stored in each of the containers using the terminology (section 2.2.1.1) of the MobJeX implementation presented in this thesis.

- ServiceMetricsContainer
  - Processor Usage (PU)
  - Network Usage (NU)
- RuntimeMetricsContainer
  - Memory Usage (MU)
- TransportManagerMetricsContainer
- MobjectMetricsContainer
- MobjectMethodMetricsContainer
  - Number of Invocations (NI)
  - Execution Time (ET)
  - Invocation Time (IT)
- ProxyMetricsContainer
- ProxyMethodMetricsContainer
  - Response Time (RT)

Note that some metrics containers do not contain any `Metric` objects, since as previously described, the sample implementation uses only the subset of global adaptation metrics from Ryan and Rossi (Ryan and Rossi, 2005). However, for the sake of consistency and to promote flexibility in terms of adding new metrics in the future, these containers are still considered in the present design.

Furthermore, as described in section 2.1.1.1 measurement is needed at various levels of granularity down to the method level. As such, this solution groups the metrics for each method in a separate container. The primary advantage of aggregating metrics in containers is that it allows them to be propagated easily as a cohesive unit through the components of the mobility framework so that they can be delivered to the adaptation engine, as discussed in the following subsection.

3.6 Propagation and Delivery of Metrics

As concluded in section 2.2, having identifying the limitations of existing work, metrics should be propagated through the components of the mobility framework so that they can be delivered in aggregate form to the adaptation engine. Consequently, this thesis identified and compared three ways of providing metrics propagation to the host manager, and determined that attaching metrics containers to an abstract model of the system rooted in the host manager was the best solution as described below.

The first alternative involved creating a metrics container which was referenceable from each of the system components such as host managers, runtimes and mobile objects. Furthermore, in order to remove the need for metrics propagation dependencies in the actual system, a containment relationship was created between metrics containers to duplicate the structural containment relationship of the system components. As such, once components were propagated from lower (e.g. mobject) to higher (e.g. host controller) level metrics containers, they would also be visible to the system component maintaining a reference to that higher level container. The disadvantage of this approach is that

whenever an object migrates to a remote host, the links between metrics containers need to be updated which is a non trivial task.

The second alternative involves attaching each metrics container to its corresponding system component and utilizing the already established system containment relationship. The problem with this approach is that metric containers are sent individually with no relationship between them. Consequently, the structural relationship between metrics is lost (for example which mobject metrics belonged to which runtime), meaning that the structure would somehow need to be reconstructed at either the host manager or adaptation engine. Again this is a non-trivial task, and potentially computationally expensive.

Due to the disadvantages of the first and second alternatives, the third approach is to maintain an abstract representation or *model* (Randell et al., 1999) of the system. Such a model would contain *model entities*, corresponding to each of the main system components, connected in a tree like hierarchy, which precisely reflects the structure and containment hierarchy of the actual system. Attaching metrics containers to model entities allows a model entity representing a host manager to be delivered to the adaptation engine enabling it to access all metrics in that component and any of its children (i.e. runtimes, and mobile objects). Furthermore it would generally be expected that an adaptation engine or system controller would already maintain a model of the system that can not only be reused for propagation but also provides an effective means of *delivering* metrics information from the host manager to the adaptation engine.

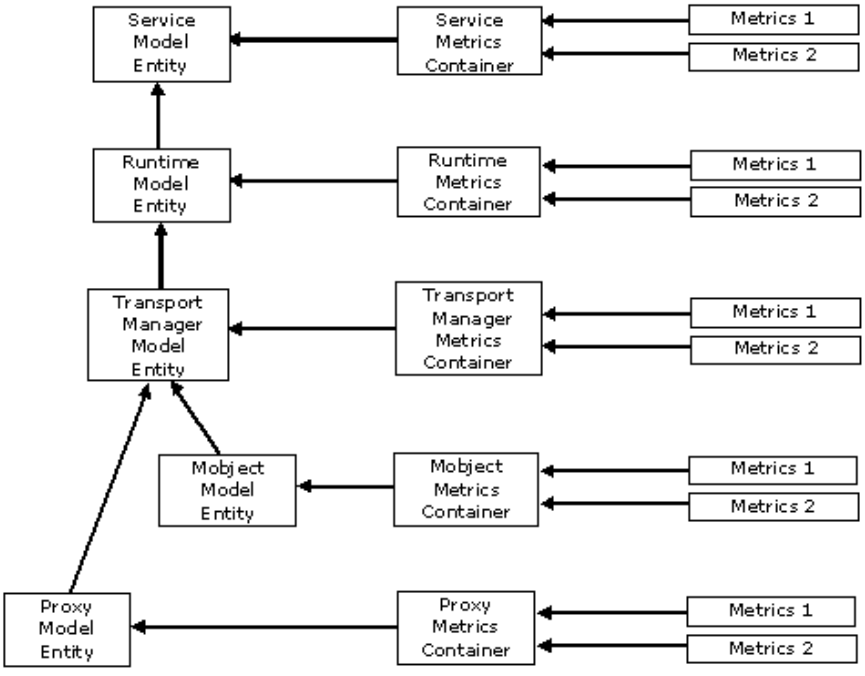

Figure 2: An illustration of the relationship between Model Entity, Metrics Container, and Metrics in the containment hierarchy

The following discussion relates to the performance considerations of propagating metrics in the MobJeX mobility subsystem that are different to other mobile object frameworks such as FarGo. In

MobJeX, the host manager (service) runs in a different process from its runtimes, with the communication between these two components done through Remote Method Invocation (RMI) (Waldo, 1998) on the same host. Therefore, even though it is not a remote call across a network, the data passed between these two components is still marshalled and unmarshalled involving the computationally expensive serialisation and deserialisation of objects. Consequently, transferring unchanged metrics unnecessarily increases overhead and as a result this thesis proposes a flexible solution for reducing the propagation frequency, and thus overhead, in section 3.7. Note that this particular type of overhead is not an issue with mobile object frameworks that do not have a separate host manager (service), because in this case, the runtime is the final destination of the metrics. However, this design ultimately causes greater total overhead, since each runtime must deliver metrics to the remote adaptation engine separately, which is even more expensive since it is a remote call across the network.

3.7 Propagation and Delivery Criteria

This subsection proposes flexible criteria to allow each component to decide when it should propagate its metrics to the next component in line (figure 3), in order to reduce the overhead incurred when metrics are unnecessarily propagated through the components of the mobility framework and delivered to the adaptation engine.

This thesis proposes four different types of criterion that are executed at various stages of the measurement and propagation process.

- *metrics update criteria*, which is used by the `Metric` object to decide when to collect metrics.
- *metrics container notify criteria* is used by the `Metric` object to decide when to notify the container that there is a metric update.
- *metrics push request criteria* is used by the `MetricsContainer` object to decide when to request a push to its parent model entity.
- *metrics push criteria* is used by the `ModelEntity` object to decide when to push metrics and what metrics to push.

These criteria are evaluated at specified points in the collection (measurement, propagation and delivery) in order to determine whether the next action should be taken or not. This approach was designed such that whenever a single criterion is not satisfied, the subsequent criteria are not tested.

In the MobJeX implementation devised for this thesis, the instantiation of criteria was performed using the Factory Method design pattern (Gamma et al., 1995), thereby allowing new criteria (i.e. different implementations) to be introduced without affecting the objects that use the criteria. This approach was used for testing simple criteria variations that are evaluated in section 4. Note that the two generic criteria implementations of *never* and *always* can be used for all different types of criterion. The *always* criterion simply returns true, which means that the action will always be performed. Conversely, the *never* criterion returns false, meaning that the action will never be performed. Depending upon the implementation platform these criteria would be inlined at compile time to negate any performance overhead in cases where criteria are deemed to be irrelevant at particular points in the collection process.

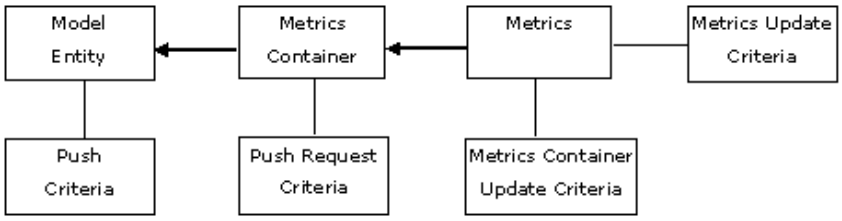

Figure 3: The different type of criteria

3.7.1 Metrics Update Criteria

This criterion is used by the `Metric` object to decide whether a new metric value needs to be stored. This has the advantage of minimizing the metrics history storage size by preventing the collection of unnecessary metrics (e.g. metrics with the same values or similar values). In doing so this avoids any further processing of the current metrics and hence the evaluation of the subsequent criteria (i.e. metrics container notify criteria, metrics push request criteria, and metrics push criteria). An example of a metrics update criteria for a resource utilisation metric such as Memory Usage (MU), would be a filter based in which the newly measured metric is only stored if it exceeds a specified difference threshold.

3.7.2 Metrics Container Notify Criteria

This criterion is based on the assumption that there may be cases where it is desirable to store a metric in the history for the analysis of temporal behaviour, but is not yet significant enough to notify the `MetricsContainer` for further processing. A simple example of this criterion is the same as the metrics update criteria above in that the newest metric value is compared with the previously stored value to determine whether the difference is significant enough to be of any interest to the `MetricsContainer`.

3.7.3 Metrics Push Request Criteria

This criterion, which belongs to the `MetricsContainer` exists because each `MetricsContainer` can have multiple `Metric` objects, of which it has explicit domain knowledge and can thus determine if, when, and how many of these metrics should be propagated to the next component.

A simple implementation to reduce the number of push requests would be waiting for a certain number of updates before sending a push request to the model entity. For example, since the `MobjectMetricsContainer` object contains three metrics, a possible criteria would be to check if more than half of the metrics have changed.

A slightly more advanced implementation can be done by giving each metric a weight to indicate how significant it is in the adaptation decision making process.

Finally, an example of a domain specific criterion for a `MobjectMethodMetricsContainer` involves considering the relationship between the three specific metrics of Number of Invocations (NI), Execution Time (ET), and Invocation Time (IT). In this case we may choose to delay issuing a push request until all three of the metrics have been received.

3.7.4 Metrics Push Criteria

The previous three criteria served only as notifications since they were local calls and could use pass by reference semantics. In contrast the metrics push criterion accommodates two types of decision making, in terms of what to push and when to push, in order to reduce both size and frequency of pushes. This decision is based on the push requests received from the metrics container, and the push notifications received from the child model entities.

Note that pushing metrics between local model entities is still computationally inexpensive since it is based on pass by reference semantics. As such, the push criterion does not become important until the runtime is reached upon which the size of pushed data should be reduced as much as possible. This is because pushing metrics through RMI (Waldo, 1998) is expensive due to the overhead of marshalling and unmarshalling the data as described earlier in section 3.6. Furthermore, although it is always important to reduce the number of pushes, this is especially important from the service to the adaptation engine because of the presence of network latency meaning that for the same amount of data, it is more efficient to send the data in an aggregate form rather than sending smaller chunks multiple times.

A simple implementation for reducing the number of pushes can be done using the concept of process period (Rolia and Lin, 1994) described in section 2.2.1.3. In this case the model entity accumulates pushes from its child entities until the process period expires at which time it pushes the accumulated metrics to its parent.

For reducing the size of pushed data, two types of pushes were considered: *shallow* push and *deep* push. In shallow push, a list of metrics containers that contain updated metrics is pushed. In deep push, the model entity itself is pushed, along with its metrics container and its child entities, which also have reference to metrics containers but possibly unchanged metrics.

4 Evaluation

In order to address research question 2, this section presents an empirical evaluation of the solution presented in the previous section. The section is structured in terms of aim, equipment, methodology and results.

4.1 Aim

The tests presented in this section had a number of aims: 1) To test the correctness of the solutions, 2) To evaluate the impact on application performance in terms of metrics collection overhead by comparing a sequence of executions with and without metrics collection, 3) To evaluate the correctness of the criteria based solution and to test the effectiveness of a number of simple criteria implementations, and 4) Finally, to show that under certain circumstances, the metrics collection overhead is low enough to facilitate adaptation for the purpose of improving overall system performance.

Note that with regard to the last aim, the maximum acceptable metrics collection overhead is dependent on both the performance requirement of an application and the adaptation strategy used. As an example, the local adaptation strategy proposed by Ryan and Rossi (Ryan and Rossi, 2005) showed up to 50% improvement in system performance and resource utilization. Therefore, it could be argued that even with a relatively high 40% metrics collection overhead, the metrics would be worth collecting in order to facilitate local adaptation for a net gain of 10% in overall application performance.

4.2 Equipment

All tests were executed using two Pentium 4 3.0 GHz PCs with 1,024 MB of RAM, running Java 1.4.2 08. The two machines were connected to a router with one other computer acting as a file server, so that the two machines shared the same application byte code. No other machines were connected to the router to avoid fluctuations in network performance that were independent of the execution scenarios under investigation. The timing tests utilised a native timing library with 10 microsecond resolution enabling more accurate measurement than the standard JVM was able to provide.

4.3 Methodology

All the tests were run under the following general conditions, unless otherwise specified below in the discussion of the individual tests.

The tests were run using a synthetic application consisting of a minimum of 2, up to a maximum of 52, mobile objects, with each object having a reference to the other. Tests were also conducted using varying numbers of methods in each object. These objects were initially created on one machine, referred as the *local* machine, with one of the objects then moved to the other machine (*remote* machine). Once the object had moved, the local object invoked a method of the remote object, thereby initiating the metrics collection process.

The evaluation of the metrics collection overhead was divided into two parts each which were separately measured: 1) Mobject Metrics Collection Overhead (MMCO), which quantified the overhead incurred with metrics collection in the object implementation, and 2) Proxy Metrics Collection Overhead (PMCO) which concerned the overhead in the proxy.

In order to measure the Mobject Metrics Collection Overhead (MMCO), code instrumentation was used to simulate various Execution Times (ET) within the method of the remote object. In order to control the length of execution time, a call to the sleep() method was inserted in the method body of the remote object to shorten or lengthen the duration spent executing the method. For the purpose of the tests described herein, the ET value was increased from 500 to 10000 milliseconds, in increments of 500 milliseconds.

MMCO was derived using the following formula:

MMCO = (ETnew - ETorig) / ETorig * 100.0

Where:

MMCO is the percentage overhead of the metrics collection process occurring in the mobject.

ETnew is the Execution Time of the method with metrics collection code

ETorig is the Execution Time of the method without metrics collection code

In order to measure the Proxy Metrics Collection Overhead (PMCO), the method of the remote object was made to accept one argument of the byte array type. The byte array is used for varying the Size of Serialised Parameters (SSP), and thus the Invocation Time (IT), since object serialization needs to be done when the array object is transferred over the network. For the purpose of the tests presented in this thesis, the SSP value was increased from 500 KB to 10000KB, in increments of 500KB.

PMCO was derived using the following formula:

PMCO = (ITnew - ITorig) / ITorig * 100.0

Where:

PMCO is the percentage overhead of the metrics collection process occurring in the proxy.

ITnew is the Invocation Time of the method with metrics collection code

ITorig is the Invocation Time of the method without metrics collection code

4.4 Limitation

Since only a limited number of tests could be executed, this evaluation chose to measure the worst case scenario in which all metrics collection was initiated in mobjects, wherein the propagation cost is higher than for any other metrics collected in the system. In addition, since exhaustive testing of criteria is beyond the scope of this thesis, only one type of criteria was used in the tests. The *metrics push request criteria* was chosen, because we believe that the propagation of metrics contributes the most to the metrics collection overhead. Therefore, some simple criteria were used in order to show their impact on application performance. Extensive testing of criteria, and how this affects specific adaptation approaches, is left to future work.

The following simple criteria were used within the tests:

- Always criterion

  This criterion uses a condition that is always satisfied, meaning that every time a metric is measured, it will be pushed (i.e. propagated) all the way to the service.

- Time based criterion

  This implementation is based on the concept of a process period, as described in section 2.2.1.3. Using this criterion, pushes will only be initiated if the last propagation was performed more than T milliseconds ago. The choice of T is discussed in section 4.6.

- Frequency based criterion

  This type of criteria keeps track of the number of potential propagations, and decides when to push metrics depending on N, the number of potential propagations that were ignored since the last propagation.

4.5 Results - Always Criteria

This subsection discusses the results of the tests that used the *always* criteria for evaluating the metrics collection overhead of the application described in section 4.3. Different numbers of methods and different numbers of objects were used in order to show how these application attributes affected the metrics collection overhead in terms of MMCO and PMCO.

4.5.1 MMCO

These tests involve 5 executions, where the first execution involves a single object with a single method. For the purpose of varying the number of methods, the two subsequent executions were done by increasing the number of methods, each time by 25. In this case, the number of remote mobjects was not changed, so that the overhead of having additional methods can be evaluated in isolation. Conversely, in the other two executions, the number of objects was varied; however each object had only one method. To summarise, the combinations used in the tests were:

- 1 mobject with 1 method
- 1 mobject with 26 methods
- 1 mobject with 51 methods
- 26 mobjects with 1 method

- 51 mobjects with 1 method

Where the source machine always had a single object initiating the execution.

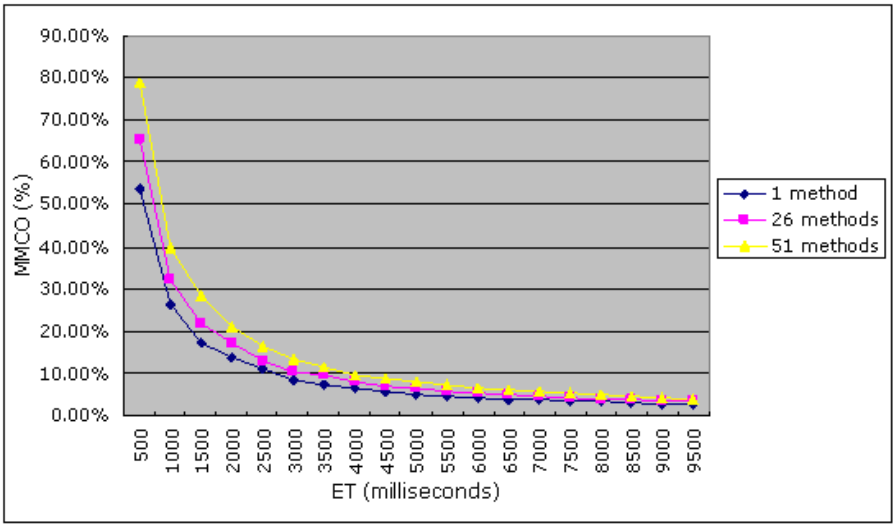

Figure 4: The use of always criteria in the measurement of MMCO with different numbers of methods

The results (figure 4) of the test with different numbers of methods shows that the number of methods in an application affects the metrics collection overhead. This is because each method has a `MobjectMethodMetricsContainer` associated with it. Therefore, the more methods an object has, the more data that is pushed during the metrics propagation to the service, which increases the overall metrics collection overhead. However, although the application performance changes as the number of methods and/or the number of objects changes, its performance characteristic is not affected, which is why all the lines in the graph have similar shapes.

Figure 4 also shows that the MMCO is relatively high when the method has low execution time, which implies that particularly in the case where no criteria is used or when always criteria are used, the collection of metrics is not desirable for methods with short execution. Different criteria can be used for reducing this overhead, thus section 4.6 and 4.7 discusses empirical tests on the impact of some simple criteria implementations on the MMCO.

The results of figure 5 show that the number of mobjects in the application also affects the metrics collection overhead, as caused by the additional `MobjectMetricsContainers` that need to be propagated to the service. It can be seen that although increasing the number of mobjects has similar effect to increasing the number of methods, the actual overhead of having more mobjects is higher than the overhead of having more methods. This is because the size of a `MobjectMetricsContainer` object is larger than a `MobjectMethodMetricsContainer`, because it is the container of the `MobjectMethodMetricsContainers`.

4.5.2 PMCO

The tests undertaken for measuring PMCO were similar to those for MMCO described in section 4.5.1, with the same 5 combinations of number of methods and number of mobjects used. However, because varying the number of methods and number of mobjects has no effect on PMCO, all 5 execution

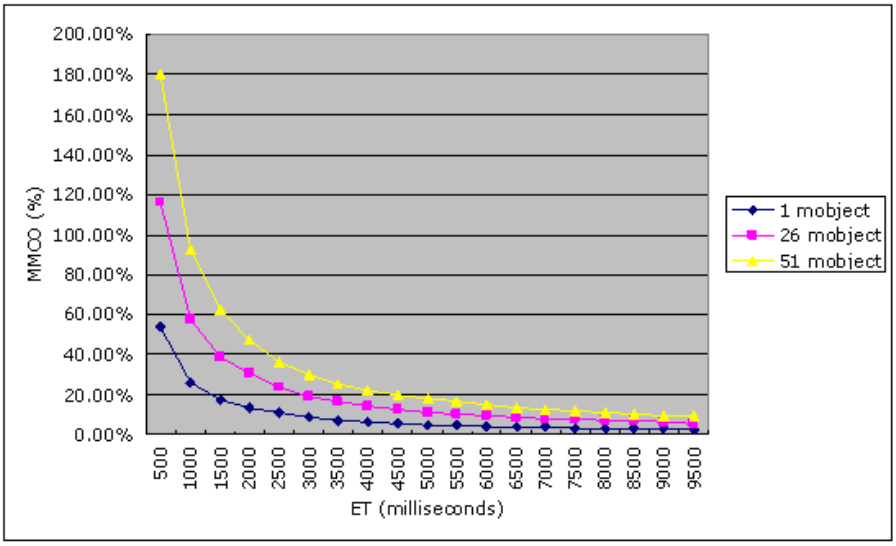

Figure 5: The use of always criteria in the measurement of MMCO with different number of objects

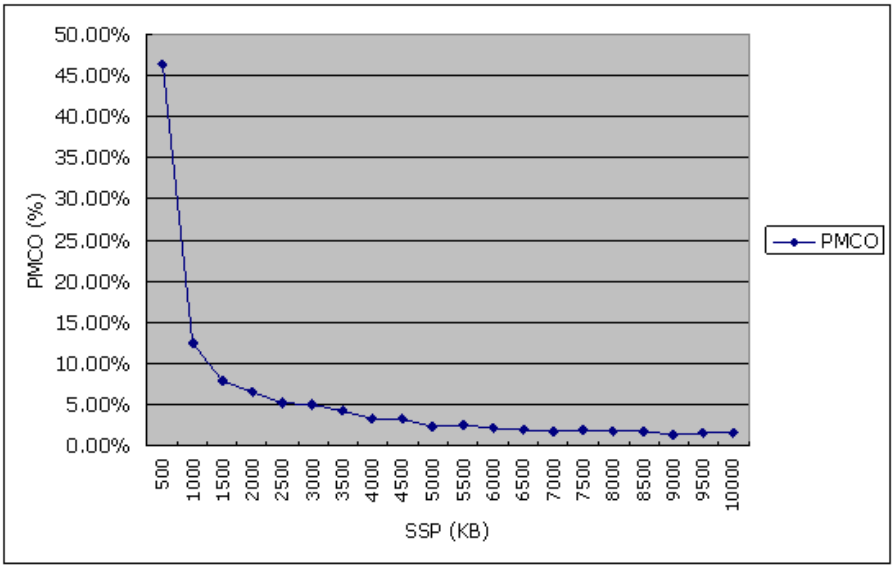

Figure 6: The use of always criteria in PMCO measurement

combinations produced the same results and thus only one PMCO graph is shown for the always criterion in figure 6.

As was the case with MMCO, PMCO is high when the SSP (and thus the Invocation Time (IT)) is low. Nevertheless, PMCO has a lower overhead than MMCO because metrics collection in the proxy only involves the measurement of metrics, whereas collection in the mobject involves both measurement and propagation.

4.6 Results - Frequency-based Criteria

This subsection discusses the tests taken to evaluate the impact of using a simple frequency-based criteria. Again the results are presented in terms of MMCO and PMCO.

4.6.1 MMCO

This test was performed by varying the value of ET as discussed previously in section 4.3. In addition, the frequency value (N) of the chosen criteria ranged between 2 and 6, since the value of 1 is effectively the same as the always criteria. The results are presented as a 3-dimensional graph in figure 7, with ET, N, and MMCO comprising the axes. Although it can be seen in figure 4 and 5 that the different number of methods and mobjects affects the performance (i.e. overhead) of the application, it does not affect its performance characteristic. Therefore, for brevity, and to concentrate on the impact of the N criteria, this test was done using the combination of 26 mobjects and 26 methods.

As expected, ignoring the N frequency, the results of the test in figure 7 show similar performance behaviour to the previous tests of figure 4 and 5. Figure 7 also shows that when ET is low, the overhead is high, however, as the value of N increases, the overhead of the metrics collection decreases. In the case where the metrics collection overhead is already low (i.e. the ET is high), increasing the N frequency does not have much impact on the metrics collection process overhead.

4.6.2 PMCO

Once more, the measurement of PMCO is similar to MMCO except that SSP was varied on the x-axis instead of ET. Consequently, the results are presented in a 3-dimensional graph in figure 8, with SSP, N, and PMCO on each axis. Again, as discussed in 4.6.1, for the sake of brevity, this test was done using 26 mobjects where each mobject has 26 methods.

As expected, the use of criteria does not have any effect on the PMCO, as reflected in the graph wherein even as N increases, the PMCO does not change. This is because in this particular test, criteria are used for metrics propagation only, and as mentioned in section 4.5.2, the propagation of metrics does not affect PMCO.

4.7 Results - Time-based Criteria

This subsection discusses the tests taken to evaluate the impact of using a simple frequency-based criteria implementation on the Mobject Metrics Collection Overhead (MMCO). Since the previous section has demonstrated that the use of criteria does not affect the PMCO, PMCO will not be discussed further in this subsection.

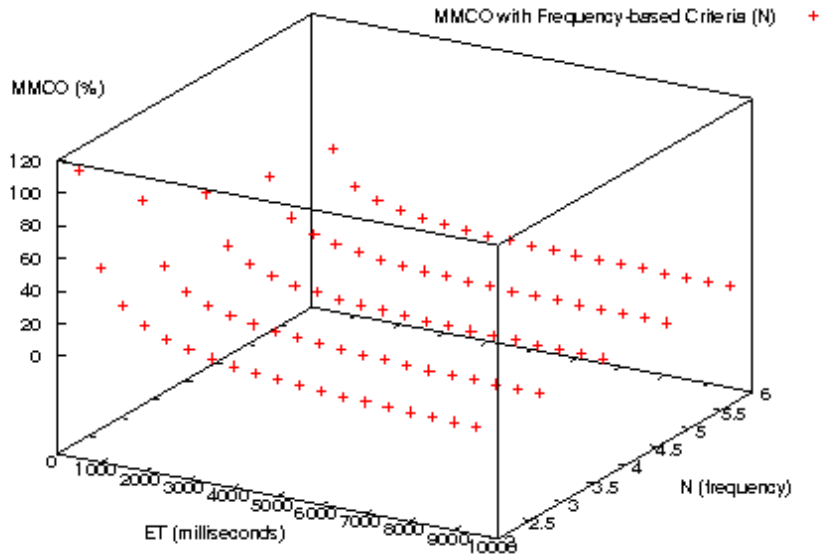

Figure 7: The use of frequency-based criteria (N) in the MMCO measurement

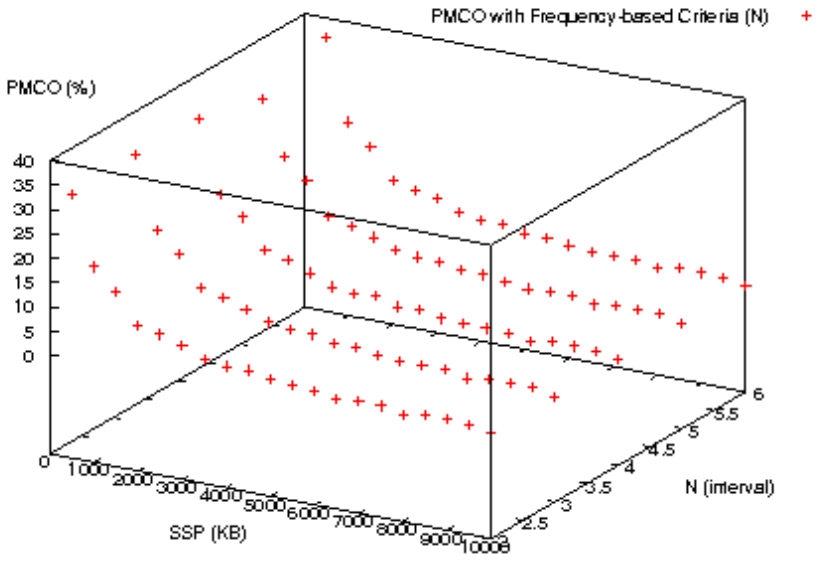

Figure 8: The use of frequency-based criteria (N) in the PMCO measurement

4.7.1 MMCO

The procedure taken to run this test is the same as the procedure mentioned in section 4.6.1, except for the criteria used. This test used a time-based criteria implementation with T values that range from 0 to 40 seconds, with a 2 second increase.

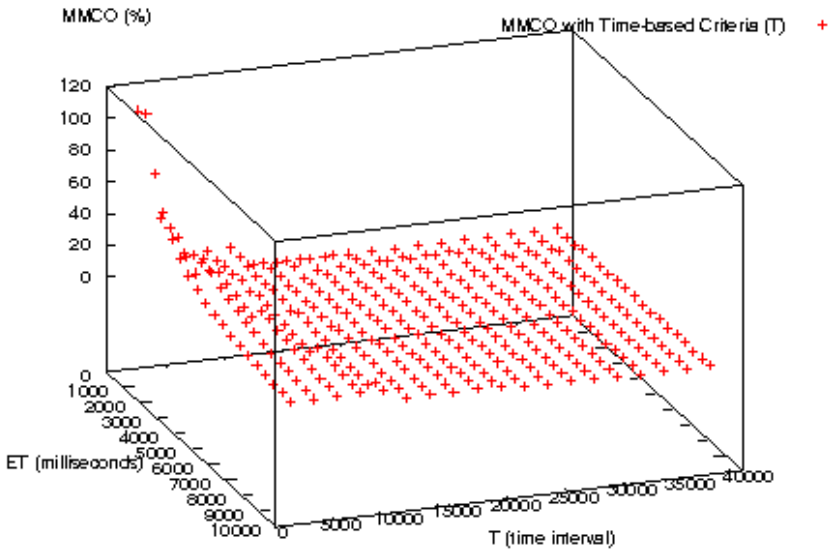

Figure 9: The use of time-based criteria (T) in the MMCO measurement

Figure 9 shows that the performance characteristics of the application are similar to the characteristics shown in figure 8, except that the metrics collection overhead in this test decreases faster. This is because some values of T are too large so that for *small* ETs, no metrics collection was performed at all. On the other hand, for *large* ETs, metrics collection overhead is always small compared to the ET, no matter how much metrics collection was performed. Therefore, it can be seen that a big portion of the graph forms a flat surface, which indicates that the metrics collection overhead has almost reached its minimum point.

5 Summary and Conclusion

The main contribution of this thesis has been to propose, test and evaluate a solution for collecting both environmental and software metrics and delivering them to an adaptation engine. Specifically, this thesis has implemented a solution based upon an abstract representation of the system, which holds containers that aggregate metrics for each specific component comprising a mobile object framework, including host mangers, runtimes and mobile objects. A key feature of the solution is the specification of multiple configurable criteria to control the propagation of metrics through the system, thereby reducing collection overhead.

A number of different criteria types and operations were defined in order to provide fine grained control over the propagation of metrics within the system and the delivery of only those metrics that are relevant to the decision making process of the adaptation engine. These criteria are 1) *metrics*

update criteria, which is used by the `Metric` object to decide when to store measured metrics. 2) *metrics container notify criteria*, used by the `Metric` object to decide when to notify the container that there is a significant metric update, 3) *metrics push request criteria*, which is used by the `MetricsContainer` to decide when to request a push from its parent model entity, and 4) *metrics push criteria*, as used by the `ModelEntity` object to decide when and what metrics to push.

The metrics of Ryan and Rossi (Ryan and Rossi, 2005) were chosen as the basis for the solution presented in this thesis, since they are specifically intended for mobile application adaptation as well as having been derived from a series of mathematical models and empirically validated; and empirically shown to improve the application performance in a real adaptation scenario compared with a random non-adapted base case.

The MobJeX (Ryan and Westhorpe, 2004) platform was used as the basis for implementation, incorporating a functional subset of the metrics described above that can support a global adaptation strategy. Furthermore, synthetic applications were evaluated in order to provide an initial performance benchmark in terms of metrics collection overhead for the subset of metrics in question. Preliminary findings demonstrate that even with simple collection criteria it should be possible to achieve positive adaptation outcomes that offset the overhead of the metrics collection process itself.

By implementing these solutions and integrating them into the MobJeX framework, this thesis directly facilitates further research on the increasingly important area of mobile application adaptation.

In particular, the design of a metrics collection and propagation strategy based on configurable criteria encourages further work in terms of the specification of more detailed criteria based on the temporal characteristics of metrics, in order to further reduce the impact of the metrics collection process and thus continue to increase the benefits in performance that can be achieved via adaptation.

In addition, future work will include the comparison of the performance of adaptation based on estimation, with that using the direct measurement of metrics. Furthermore, additional metrics will be explored, including, but not limited to, metrics related to quality and coupling.

Acknowledgements

I would like to thank my supervisor Dr Caspar Ryan for his introduction to MobJeX during my summer studentship within the ATcrc, his support during the year, and especially for all of his proof reading near the end. I would also like to thank Dr Pablo Rossi, for all his help (especially on metrics and adaptation) and encouragement this semester.

References

Arisholm, E., Briand, L. C., and Føyen, A. (2004). Dynamic coupling measurement for object-oriented software. *IEEE Transactions on Software Engineering (TSE)*, 30(8):491–506.

Chen, G., Kang, B.-T., Kandemir, M. T., Vijaykrishnan, N., Irwin, M. J., and Chandramouli, R. (2004). Studying energy trade offs in offloading computation/compilation in Java-enabled mobile devices. *IEEE Transactions on Parallel and Distributed Systems*, 15(9):795–809.

da Silva e Silva, F. J., Endler, M., and Kon, F. (2003). Developing adaptive distributed applications: A framework overview and experimental results. In *Proceedings of the International Symposium on Distributed Objects and Applications CoopIS/DOA/ODBASE*, pages 1275–1291.

Farkas, K. I., Flinn, J., Back, G., Grunwald, D., and Anderson, J.-A. M. (2000). Quantifying the energy consumption of a pocket computer and a Java virtual machine. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (ACM SIGMETRICS)*, pages 252–263, Santa Clara, CA.

Flinn, J. and Satyanarayanan, M. (1999). Energy-aware adaptation for mobile applications. In *Proceedings of the 17th ACM Symposium on Operating Systems and Principles SOSP*, pages 48–63, Kiawah Island.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns – Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley.

Gazit, H., Ben-Shaul, I., and Holder, O. (2000). Monitoring-based dynamic relocation of components in fargo. In *ASA/MA*, pages 221–234.

Hirschfeld, R. and Kawamura, K. (2004). Dynamic service adaptation. In *ICDCS Workshops'04*, pages 290–297.

Holder, O., Ben-Shaul, I., and Gazit, H. (1999). System support for dynamic layout of distributed applications. In *19th IEEE International Conference on Distributed Computing Systems (ICDCS '99)*, pages 403–413, Washington - Brussels - Tokyo. IEEE.

Jing, Helal, and Elmagarmid (1999). Client-server computing in mobile environments. *CSURV: Computing Surveys*, 31(2):118–157.

Katz, R. H. (1994). Adaptation and mobility in wireless information systems. *IEEE Personal Communications*, 1:6–17.

Lemlouma, T. and Layaïda, N. (2004). Context-aware adaptation for mobile devices. In *Mobile Data Management*, page 106.

Lindholm, T. and Yellin, F. (1996). *The Java Virtual Machine Specification*. Addison-Wesley.

McGregor, J. D., Cho, I.-H., Malloy, B. A., Curry, E. L., and Hobatr, C. (1999). Collecting metrics for corba-based distributed systems. In *Empirical Software Engineering*, volume 4, pages 217–240.

Microsoft (2005). .NET. http://msdn.microsoft.com/netframework/.

Morasca, S. (1999). Measuring attributes of concurrent software specifications in petri nets. In *Proceedings of International Software Metrics Symposium IEEE METRICS*, pages 100–110.

Narayanan, D., Flinn, J., and Satyanarayanan, M. (2000). Using history to improve mobile application adaptation. In *Proceedings of the 3rd IEEE Workshop on Mobile Computing Systems and Applications WMCSA*, page 31.

Noble, B., Satyanarayanan, M., Narayanan, D., Tilton, J. E., Flinn, J., and Walker, K. R. (1997). Agile application-aware adaptation for mobility. In *Proceedings of the 16th ACM Symposium on Operating Systems and Principles SOSP*, pages 276–287, Saint-Malo, France.

Philippsen, M. and Zenger, M. (1997). JavaParty - transparent remote objects in Java. *Concurrency - Practice and Experience*, 9(11):1225–1242.

Randell, L. G., Holst, L. G., and Bolmsjö, G. S. (1999). Incremental system development of large discrete-event simulation models. In *Winter Simulation Conference*, pages 561–568.

Rolia, J. and Lin, B. (1994). Consistency issues in distributed application performance metrics. In *Proceedings of the 1994 conference of the Centre for Advanced Studies on Collaborative research*, page 62.

Rossi, P. and Fernandez, G. (2003). Definition and validation of design metrics for distributed applications. In *Ninth International Software Metrics Symposium IEEE METRICS 2003*, page 124, Sydney.

Rossi, P. and Ryan, C. (2005). An empirical evaluation of dynamic local adaptation for distributed mobile applications. In *Proceedings of the International Symposium on Distributed Objects and Applications (DOA 2005)*, pages 537–546, Larnaca, Cyprus.

Ryan, C. and Rossi, P. (2005). Software, performance and resource utilisation metrics for context-aware mobile applications. In *Proceedings of International Software Metrics Symposium IEEE METRICS 2005*, Como, Italy.

Ryan, C. and Westhorpe, C. (2004). Application adaptation through transparent and portable object mobility in Java. In *Proceedings of 2004 International Symposium on Distributed Objects and Applications CoopIS/DOA/ODBASE (2)*, pages 1262–1284, Catania, Italy.

Shapiro, M. (1986). Structure and encapsulation in distributed systems: The proxy principle. In *ICDCS*, pages 198–204.

Sun Microsystems (2005). Java. http://java.sun.com/.

Voyager (2005). Voyager. http://www.recursionsw.com/voyager.htm.

Waldo, J. (1998). Remote procedure calls and Java remote method. *Invocation. IEEE Concurrency*, 6(3):5–7.

A Appendix A

Attribute	Definition	Metric	Unit
Software			
Object Compilation Volume	The size of an executable module (e.g. Java .class file)	Executable Code Size (ECS)	byte
Object Serialisation Volume	The size of a serialised object	Serialised Object Size (SOS)	byte
Object Memory Volume	The size of an in-memory object	Object Memory Size (OMS)	byte
Method Execution Volume	The size of the extra memory required during the execution of a method	Execution Memory Size (EMS)	byte
Method Body Intensity	The processing intensity of a method in terms of executable statements/instructions	Number of Executed Instructions (NEI)	int
Method Interface Volume	The combined size of the parameters of a method interface	Size of Serialised Parameters (SSP)	byte
Method Invocation Frequency	The rate of occurrence of method invocation	Number of Invocations (NI)	int
Performance			
Method Execution Cost	The execution cost of a method, ignoring any overhead associated with call semantics	Method Execution Time (ET)	ms
Method Invocation Cost	The cost of calling a method, independent of its actual processing e.g. marshalling etc.	Method Invocation Time (IT)	ms
Object Migration Cost	The cost of moving an object instance (e.g. serialised Java object) between hosts	Migrate Instance Time (MIT)	ms
Class Migration Cost	The cost of moving a class implementation (e.g. Java .class file) between hosts	Migrate Class Time (MCT)	ms
Resource Utilisation			
Network	The network resource available between two hosts	Network Capacity (NC)	byte/s
Network Utilisation	The aggregate network bandwidth between two hosts	Network Usage (NU)	byte
Memory	The total memory available on a host	Memory Capacity (MC)	byte
Memory Utilisation	The aggregate memory usage of a host	Memory Usage (MU)	byte
Processor	The processing power of a host	Processor Capacity (PC)	int/s
Processor Utilisation	The aggregate processor usage of a host	Processor Usage (PU)	int

Figure 10: Software, Performance, and Resource Utilisation Attributes and Metrics. (Taken from Ryan and Rossi 2005)