A Distributed Approach to Local Adaptation Decision Making for Sequential Applications in Pervasive Environments

Ermyas Abebe and Caspar Ryan

School of Computer Science & IT, RMIT University,
Melbourne, Victoria, Australia
{ermyas.abebe,caspar.ryan}@rmit.edu.au

Abstract. The use of adaptive object migration strategies, to enable the execution of computationally heavy applications in pervasive computing spaces requires improvements in the efficiency and scalability of existing local adaptation algorithms. The paper proposes a distributed approach to local adaptation which reduces the need to communicate collaboration metrics, and allows for the partial distribution of adaptation decision making. The algorithm's network and memory utilization is mathematically modeled and compared to an existing approach. It is shown that under small collaboration sizes, the existing algorithm could provide up to 30% less network overheads while under large collaboration sizes the proposed approach can provide over 900% less network consumption. It is also shown that the memory complexity of the algorithm is linear in contrast to the exponential complexity of the existing approach.

1 Introduction

With increasingly capable devices emerging in mobile computing markets and high speed mobile Internet permeating into current infrastructure, there is a growing need to extricate complex computing tasks from stationary execution domains and integrate them into the growing mobility of day to day life. Given the computationally heavy nature of many existing software applications, such a transition requires either the creation of multiple application versions for devices with different capabilities, or for mobile computing devices, such as smartphones and PDAs, to possess the resource capabilities of typical desktop computers. Despite the increasing capabilities being built into mobile devices, the majority of these pervasive computing machines are still incapable of handling the intensive resource demands of existing applications, and it is likely that the gap between mobile and fixed device capability will remain as both types of hardware improve. Therefore, an effective way to bridge the gap between computing resource availability is for applications to adapt dynamically to their environment.

Application Adaptation refers to the ability of an application to alter its behavior at runtime to better operate in a given execution scenario. For instance, an application might reduce its network communication in the event of an increase in bandwidth cost or a detected drop in bandwidth [1] or it might spread out its memory consumption across multiple nodes due to lack of resources [2-6].

R. Meersman, T. Dillon, and P. Herrero (Eds.): OTM 2009, Part I, LNCS 5870, pp. 773–789, 2009.
© Springer-Verlag Berlin Heidelberg 2009

Adaptive object migration [2-6] is one such strategy in which a client distributes application objects to one or more remote nodes for execution. The approach allows constrained devices to execute applications with resource requirements exceeding their capabilities. Furthermore, the strategy allows the client to obtain improved performance and extended application functionality by utilizing externally available computing resources.

Adaptation entails incumbent overheads caused by the need to monitor and communicate available resources and application behavior. Additionally the computation of the runtime placement of objects or object clusters to suitable nodes incurs resource costs that could outweigh potential gains. These overheads limit the applicability of existing adaptation approaches in pervasive space. While the constrained nature of the devices in pervasive environments require that an adaptation process compute optimal decisions with minimal computation resources, the heterogeneity and indeterminate size of the collaboration, require that the solution scale adequately to diverse collaboration environments and application behavior.

This paper proposes a novel distributed adaptation decision making algorithm to improve the *efficiency* and *scalability* of adaptive object migration. The most relevant work is then compared in terms of its network and memory utilization using mathematical modeling and simulation. The results show that the proposed approach can provide more than 900% less network overhead under large collaboration sizes while maintaining linear memory complexity, compared with the exponential complexity of the existing approach. Finally, the paper outlines future work in terms of implementing and empirically evaluating the optimality of the adaptation decisions themselves.

The rest of this paper is organized as follows: Section 2 assesses existing work on adaptive object migration and identifies the most relevant work in the context of pervasive environments. Section 3 discusses the proposed distributed adaptation decision making algorithm. Section 4 mathematically models and comparatively evaluates the proposed approach to existing work. And finally Section 5 provides a summary and conclusion and outlines future work.

2 Literature Review

Object migration as a form of code mobility is not new. Early work [7-10] focused on manual migration of threads to remote nodes for objectives such as application behavior extension, load balancing and performance improvement. In contrast, the dynamic placement of objects in response to environmental changes or application requirements has been a focus of more recent research.

Adaptive object migration refers to the reactive and autonomous placement of objects on remote nodes based on a runtime awareness of the collaboration environment and application behavior. The approach allows applications to effectively utilize external computing resources to improve performance and extend application behavior while enabling devices to run applications with resource demands exceeding their capabilities.

Works on adaptive object migration differentiate between parallel and sequential applications. Adaptation for parallel applications [2, 5, 11] has typically focused on cluster and grid computing environments with objectives such as load balancing, performance improvement and improved data locality. Adaptive placement in parallel

applications generally involves the co-location of related threads or *activities*, to minimize remote calls, and the distribution of unrelated ones to reduce resource contention. Work on sequential application adaptation has focused on more diverse computing environments including pervasive spaces. With objectives such as performance improvement and load balancing, decision making in sequential application adaptation involves matching objects to available computing nodes while minimizing inter-object network communication and improving overall utility. Sequential application adaptation [3, 4, 6, 12] presents more challenges as there is less explicit division in the units of distribution. As most work on adaptation in pervasive spaces involves sequential applications, this paper identifies relevant work in that domain alone. Nevertheless, although the algorithm is proposed in the context of sequential application adaptation, it is expected that the concepts of distributed decision making would also be applicable for parallel application adaptation.

Adaptive Offloading techniques [6, 12] and the approach presented by Rossi and Ryan [3] are works on adaptation for sequential applications specifically targeting mobile environments. *Adaptive offloading* techniques generally assume the existence of a single mobile device and a dedicated unconstrained surrogate node to serve as an offloading target. Adaptations are unidirectional in which computation is offloaded from the constrained device to the surrogate node. The approach presented by Rossi and Ryan [3] discusses adaptation occurring within *peer-to-peer* environments in which any node could potentially be constrained at some point during the collaboration. The approach presented by Rossi and Ryan [3] is more applicable to pervasive spaces in which spontaneous collaborations are formed from heterogeneous devices. Our adaptation algorithm will hence be presented as an extension to the Rossi and Ryan [3] algorithm. The paper hereon refers to the algorithm presented by Rossi and Ryan as the *existing* algorithm and the proposed approach as the *distributed* algorithm.

2.1 Adaptation Process

Generally, application adaptation approaches involve 3 basic components: 1. a metrics collection component, responsible for monitoring application behavior and available computing resources within the collaboration, 2. a decision making component responsible for determining object-to-node placements based on information from the metrics collection component and 3. an object migration component responsible for moving selected objects to their designated targets.

Based on the site at which decision making occurs, Ryan and Rossi [13], identify two types of adaptation processes, *Global (centralized)* and *Local (decentralized)* adaptation.

Global Adaptation: In *Global or Centralized Adaptation,* decision making is performed by a single dedicated and unconstrained machine. Other nodes within the collaboration periodically communicate their environment and application metrics [14] to this central node. The metrics pushed out by a node, includes the resource usage measurements of the device and the individual objects in its address space. When a node within the collaboration runs out of resources, the central node computes a new object distribution topology for the collaboration and offloads computation from the constrained device. The object topology is computed with the objective of providing optimum load balance and performance improvement while minimizing inter-object network communication.

While *Global Adaptation* allows for the near optimal placements of objects-to-nodes, the computation costs of computing an object topology for even a simple scenario with a few objects in small collaborations can be prohibitively expensive [13]. While Ryan and Rossi [13] discuss the possible use of Genetic Algorithms as a solution to reduce this cost, the approach would still be computationally expensive compared to a decentralized approach to decision making *(discussed in the next section)*. Another disadvantage of *Global Adaptation* is the need for a reliable and unconstrained node for decision making. This presents a central point of failure and limits its applicability within ad-hoc collaborations of constrained devices.

Local Adaptation: In *Local or Decentralized Adaptation*, decision making is computed on individual nodes. Resource metrics of each node are periodically communicated to every other node within the collaboration. Unlike Global Adaptation, the metrics propagation includes only the collaboration's resource availability and not the software metrics of objects [14] (e.g. number of method invocations, method response times etc.). When a node runs out of resources, it computes an adaptation decision based on the information it maintains about the collaboration and the metrics of the objects in its memory space.

As the adaptation decisions are computed by considering only a subset of the overall object interactions, the decisions made are not as optimal as the centralized approach. However, such an approach removes the central point of failure, and offers a more scalable approach to adaptation decision making.

Figure 1 shows pseudo-code for the basic decision making computation of a local adaptation algorithm computed on individual machines, as presented by Rossi and Ryan [3]. The *evaluate* function computes a score determining the suitability of placing each mobile object, *o* on each remote target node, *n*. The object-to-node match with the highest score is selected for migration. The process is repeated until either all objects are migrated or an object-to-node match that can achieve a minimum threshold is not available. The score of an object placement (placement of object *o* to node *n*) considers two objectives: *resource offloading* from the constrained node, and *performance improvement* of the application. The *resource offloading* score evaluates the degree to which the source node's load can be mitigated and the load difference with the target node reduced whereas the *performance* score evaluates the degree of response time improvement achievable through the migration of an object to a target. Note that this serves as a basic example which can be readily extended to include more diverse goals such as battery life preservation, reliability etc.

In the context of pervasive spaces the local adaptation algorithm in Figure 1, presents a few shortcomings. As resource availability is inherently dynamic, suboptimal decisions could be made based on out-of-date information about the collaboration; therefore avoiding this problem requires nodes to communicate their environment metrics more frequently. The $O(N^2)$ message complexity[1], *where N is the number of nodes within the collaboration*, for one collaboration wide communication, increases the network overheads and limits the scalability of the approach. The storage and maintenance of this information also requires additional processor and memory resources.

---

[1] Message Complexity, in this context, is based on the number of messages sent and received by each node within the collaboration.

```
do
{
     maxScore = 0.5
     maxObject = null, maxNode = null
     for each mobile object o in local node do
       for each remote node n do
         score = evaluate(o, n)
         if (score > maxScore) then
            maxScore = score
            maxObject = o
            maxNode = n
         end if
       end for
     end for
     if (maxScore > 0.5) then
      move maxObject to maxNode
     end if
}
while (maxScore > 0.5)
```

Fig. 1. Local Adaptation Algorithm basic flow, Rossi and Ryan [3]

With a worst case complexity of $O(NM^2)$, *where M is the number of mobile objects*, decision making is computationally expensive for mobile devices in this environment, exacerbated by the decision being made on the already constrained adapting node. In response, nodes need to set lower constraint thresholds for triggering adaptation so that enough resources to compute adaptation decisions are reserved. This lowered threshold in turn results in increased adaptation throughout the collaboration causing additional network overheads and reduced application performance.

Given the diversity of the pervasive environment, adaptation algorithms need to consider additional metrics, such as location, battery usage, reliability etc. This requires adapting nodes to apply different score computation logic for different devices based on the metrics relevant to each target node, again introducing additional computation complexities to the adaptation logic.

In summary, while the approach is feasible in small, stable, homogenous collaborations, the above constraints limit the scalability under medium to large scale, heterogeneous and dynamic pervasive collaborations.

3 Distributed Approach

This section proposes a distributed decision making algorithm to improve the *efficiency* and *scalability* of the *existing* local adaptation algorithm discussed in section 2. The new *distributed* approach presents an alternative that avoids the need to periodically communicate environmental metrics throughout the collaboration, and partially distributes the decision making process to reduce the associated computation costs on the adapting node. Furthermore, the algorithm reduces the network and memory utilization costs that arise from increased collaboration size and heterogeneity.

The approach involves each node connecting to a multicast address through which all adaptation related communication will take place. The use of multicast groups for communication allows nodes to delegate the responsibility of maintaining an awareness of the collaborating nodes to external network devices (e.g. routers, switches). Once connected, each node monitors its own environment metrics and the metrics of the objects within its memory space, however unlike existing approaches this information is not communicated to other nodes. The anatomy of the *distributed* approach is described below and illustrated in Figure 2:

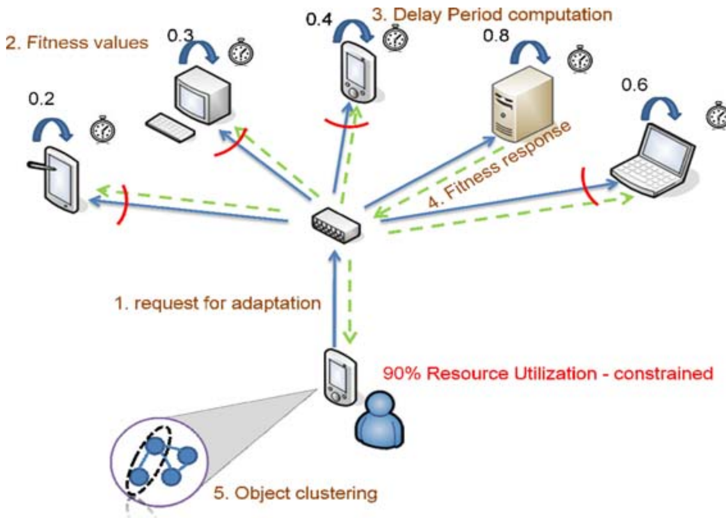

Fig. 2. Distributed adaptation decision making algorithm

1. Adaptation Request: When a node is resource constrained it multicasts an *adaptation request, R,* containing its own environment metrics and synoptic information about the objects in its memory space. The message complexity is *O(N),* since a single request is sent and *N-1* messages are received. The request sent by the node includes the following information:

- *Node URI:* This includes the IP address of the node and the port number of the middleware runtime.
- *Environmental Metrics (E):* This includes the memory, processor, network utilization and capacity of the source node (adapting node).
- *Synoptic information about objects:* The least resource utilization consumed by an object $ru^{O_{\min}}$, and the total resource utilization consumed by all mobile objects within the node, $ru^{O_{\text{total}}}$ are recorded for each metric type (memory, network, and processor).
- *Minimum Threshold Value, k:* The minimum score that can result in migration. (similar to the score threshold identified by Rossi and Ryan [3])

2. Fitness Value Computation: Each target node receives the adaptation request and determines its own suitability for adaptation by computing a *fitness score*. This *fitness score* represents the amount of resources the source can offload to the target, thereby reducing the load disparity between the two nodes; as well as the response time improvement the target can provide. Additionally, candidate nodes which are already constrained could choose not to compute adaptation decisions hence reducing the amount of global processor cycles consumed for an adaptation. The fitness value computation is further discussed in section 3.1.

3. Delay Period Computation: Each node compares its *fitness score* against the minimum threshold, *k,* sent by the source. If a node can offer a better score, it computes a delay period which is inversely proportional to its score, providing fitter nodes with shorter delay periods. The node would then wait for the computed time period before reporting its fitness score to the source. The benefit of this approach is two-fold: firstly, it reduces the possibility of a multicast *sender-storm* in which the source might be inundated by fitness responses; and secondly, it allows for the suppression of further fitness reports once the fittest candidate has responded *(see 4. below)*, thereby reducing network costs. Though such a delay scheme would introduce latency in the decision making process, it occurs parallel to the application's execution and hence is unlikely to reduce application performance. Note that since the evaluation of the *distributed* adaptation algorithm presented in this paper involves mathematical modeling and simulation, the implementation specifics of the delay period computation are not explored. Nevertheless, although there is no *prima facie* reason to assume such an implementation would be problematic, the comparative network utilization simulation in section 4 considers both cases in which a delay period is, and is not, used.

4. Reporting the Fitness Score: Once the delay period of a candidate expires, it multicasts a response to the collaboration. The response is received by the source, and every other candidate within the collaboration. Each candidate compares the offered fitness score against its own and if the offered fitness value is greater, the candidate cancels its delay timer and need not reply to the source. However, if the target node can offer a better fitness value, it will continue to wait for its delay period to expire before communicating its fitness score in the same procedure, thus giving precedence to any other node which could be *fitter* than it. Enforcing the delay timer on every response, assures that the source receives responses from only the fittest candidate(s), hence minimizing resource costs and avoiding a *sender-storm* problem. The response that is multicast by a candidate includes the following information:

- *Node URI:* This includes the IP address of the candidate and the port number of its middleware runtime.
- *Environment Metrics(E):* The capacity and utilization of each resource metric type (memory, processor, network) of the target node
- *Fitness Score, (S):* The fitness score computed by the target node, which is a value between 0 and 1. (discussed in section 3.1)
- *Required Object Metrics Value, (ru^O):* The ru^O is a value between the $ru^{O_{min}}$ and $ru^{O_{total}}$ value sent by the source (*step 1)*, which describes the resource utilization of an imaginary object cluster for which the specified fitness score can be achieved *(discussed further in section 3.1).* The source node will group objects so that the overall resource consumption of the group matches this specified value *(discussed below).*

5. Clustering of Objects: The source node listens to fitness score multicasts for a pre-computed period of time. This *wait* period is based on the amount of time it would take a node achieving the minimum threshold score *k (discussed in 1 above)* to respond. If multiple fitness scores have been received by the time the source's *wait* period expires, the node selects the best offered score. The source node then groups objects within its memory space so as to meet the criteria, ru^O, required by the selected candidate. The identified object cluster would be a subset of the mobile objects within the memory space of the constrained source node. Once the grouping of objects is complete, the source migrates the cluster to the candidate node. The migration of object clusters instead of individual objects reduces object spread, and decreases inter-object network communication cost, thereby improving the optimality of adaptive decision making when compared to the *existing* approach. As clustering would be done to meet a single constraint or resource type *(discussed in section 3.1)*, a linear computation cost of $O(M)$ can be assumed in which objects are incrementally grouped, based on sorted metrics values. Another approach to object clustering could be to use runtime partitioning approaches [12]. However, since the main focus of this paper is to compare the network and memory consumption of the new algorithm to an existing approach, the processing complexities of object clustering techniques is beyond the scope of this paper.

3.1 Fitness Score Computation

The fitness score is at that heart of the *distributed* algorithm. It determines the degree of suitability of each candidate to the adaptation request, and guides the clustering performed on the source node. Like the algorithm proposed by Rossi and Ryan [3] the computation considers two objectives: *resource offloading* from the constrained node, and *performance improvement* of the application. The *resource offloading* score for each metric type, evaluates the degree to which the load of the source node can be mitigated and the load disparity with the target node reduced. The *performance improvement* score evaluates the degree of response time improvement that can be attained by the migration. The objectives considered can be easily extended to include more diverse goals such as battery life preservation, reliability etc. Though the details of fitness computation are a part of the algorithm's *effectiveness* and hence beyond the scope of this paper, as a proof of concept, the approaches and mathematical derivations of the computation of *resource offloading* scores for each metric type (memory, processor, network) have been presented as an online appendix to this paper [15]. While the *existing* approach computes a *resource offloading* score for individual objects, the *distributed* approach has the flexibility of attaining higher scores by grouping mobile objects together. The candidate computes an ideal ru^O value, which is the resource utilization of a hypothetical object cluster for which it can offer the most ideal score for each metric type. The individual scores achieved for each metric are then aggregated to compute a final *fitness score* as is shown in the online appendix in [15].

By allowing candidate nodes to compute their own fitness value, the approach removes the need to periodically communicate environmental metrics. This reduces network communication costs and limits other resource overheads associated with storing and maintaining collaboration information on every node. Consequently, the resource utilization of the *distributed* approach is linear to the number of nodes

(discussed further in section 4), compared to the exponential cost of the *existing* approach, making it more scalable with regard to collaboration size.

By dividing and offloading the tasks of the adaptation process, the space and time complexity of decision making is reduced, making adaptation more feasible and efficient for the local constrained machine. This distribution further reduces resource contention between the executing application and the adaptation engine. Additionally, application objects are able to execute locally until higher constraint thresholds are reached, avoiding the need to maintain low thresholds so as to reserve resources for adaptation computation as is the case in the *existing* approach. Consequently this higher threshold limit reduces the number of adaptations and adaptation related overheads within the collaboration. The delegation of score computation to the remote nodes also allows individual nodes to easily factor in additional metrics into the decision computation process without further overheads on the adapting node. For instance a mobile device within the collaboration could factor in its battery life when computing a fitness score. This allows for increased diversity in the collaboration environment, as would be necessary for adaptation in pervasive environments.

As adaptive decisions are made using *live metrics* as opposed to *cached metrics*, the possibility of making suboptimal adaptation based on out-of-date metrics is minimized. The approach also lends itself to future work involving object clustering based on coupling information to obtain more efficient object topologies.

It is worth noting however, that the *distributed* approach could result in longer adaptation decision making times as a result of network latency and the delay scheme discussed in section 3. However, since adaptation decisions are performed parallel to the executing application, it is not expected to have a direct effect on the application's performance. Furthermore, it is expected that for large scale collaborations adapting computationally heavy applications, the execution of adaptation decisions on a local constrained device might take longer than the *distributed* approach which leverages externally available resources. This however relates to the performance aspect of our algorithm and is beyond the scope of this paper.

4 Evaluation

The utility of an adaptation algorithm is determined by two factors: The *efficiency* (resource utilization and performance) of the adaptation process and the *effectiveness* of the adaptation decision outcome or the optimality of the adapted object topology. Evaluating the latter requires that different application and collaboration behaviors be considered. This is because of the difference in granularity for which adaptation decisions are made and the difference in decision making location of the two algorithms. Hence an empirical study to determine the difference in decision optimality of the two algorithms would require the use of various application benchmark suites and the use of multiple collaboration environment settings. Though preliminary results of such a study are positive, further discussion of the *effectiveness* of the algorithms is left to future work.

In order to provide a comparative evaluation of both algorithms with respect to network and memory utilization, the algorithms are mathematically modeled under various environmental scenarios. Both the maximum degree with which one algorithm outperforms the other, as well as the specific environmental scenarios for which each algorithm provides comparatively lower resource utilization, are identified.

4.1 Environmental Settings

In order to compare the resource utilization of both algorithms under diverse environmental settings, a range of possible values for the variables identified in Equation 1-4 are identified and their feasibility constraints and relationships discussed.

In order to evaluate the resource utilization of the algorithms independently from their adaptation outcome, the adaptive decisions made by both algorithms are assumed to be the same. This means that both algorithms would compute the same adaptive object placements and hence incur similar object migration and inter-object communication costs under the same collaboration settings.

Table 1. Variables influencing network utilization and memory utilization in both algorithms

Variables	Constraints
Number of nodes *(N)*	$4 \leq N \leq 50$
Execution Time *(ET)*	$600s \leq ET \leq 86400s$
Frequency of propagation *(f)*	$0.0017/_s \leq f \leq 0.1/_s$
Number of adaptations *(Na)*	$Na \leq Np$
Number of propagations *(Np)*	$1 \leq Np \leq 8640, Np = ET * f$
Number of fitness values *(Nf)*	$Nf \leq N - 1$
Fitness report size *(F)*	$F = 2E$
Environment metrics size *(E)*	$200Bytes \leq E \leq 10000Bytes$
Adaptation request size *(R)*	$R = 2.5E$

Number of Nodes *(N)*: In order to observe the scalability of both algorithms with regards to collaboration size, a range of possible collaboration sizes ranging from a small scale collaboration of 4 nodes, to a large scale collaboration of 50 nodes, are considered.

Execution Time *(ET)*: The overall time spent executing and adapting a given application, ranging from a brief collaboration of 10 minutes up to 24 hours.

Frequency of Metrics Propagation *(f)*: This variable is applicable to the *existing* algorithm and refers to the frequency of propagating environmental metrics throughout the collaboration. A periodic metrics propagation is assumed for simplicity, though node triggered propagations based on degree of metrics change could be used [14]. The greater the fluctuation in resource availability within an environment the more frequently metrics propagations needs to occur. The higher value of 0.1/s (propagation every 10 seconds) would be more applicable to dynamic heterogeneous environments.

Number of Metrics Propagations *(Np= ET*f)*: The number of times a collaboration wide metrics communication occurs in the *existing* algorithm is the product of the *Execution Time (ET)* and the propagation *frequency (f)*. High number of propagations means either long executing applications, high frequency of metrics propagation or both. Hence a high number of propagations would be expected when executing long running applications in dynamic environments.

Number of Adaptations ***(Na)*****:** The number of adaptations cannot exceed the total number of metrics propagations that occur within a given period. This is because the event in which a node exceeds its resource utilization constraint thresholds, hence requiring an adaptation, would be a less frequent occurrence than a metrics propagation which reports resource changes of much lesser degree. The reason behind this theoretical upper bound is that for an adaptation to occur in the existing algorithm, the adapting node would need to first have information about the collaboration. This implies that at least one collaboration wide propagation needs to occur prior to any adaptation. As every adaptation would cause notable resource utilization differences within the environment, it needs to be followed by a subsequent propagation. This means that for every adaptation there would be at least one more number of propagations. As the adaptation decisions and number of adaptations performed by both algorithms are assumed to be the same in this evaluation, it follows that the upper bound of *Na* for both algorithms would be *Np*.

Number of Fitness Reports Returned ***(Nf)*****:** While the number of fitness values returned for each adaptation request could vary, for simplicity we assume a single *Nf* value in which a fixed number of nodes reply to every adaptation request. We consider a theoretical upper bound in which *Nf* is equal to the number of candidates, for a situation in which no functional delay scheme exists. The delay scheme, discussed in section 2, would significantly reduce the number of fitness responses by allowing only the fittest nodes to respond.

Fitness Report Message Size ***(F)*****:** As discussed in section 3, the report communicated to candidates consists of more information than the simple environment metrics, E, propagated by the *existing* algorithm. Specifically, F is two times the message size of the *existing* algorithm.

Adaptation Request Message ***(R)*****:** Similarly, it is determined that the adaptation request is 2.5 times the environmental metrics message size of the *existing* algorithm.

4.2 Network Utilization

The cost complexity of network utilization is assumed to be the total number of bytes sent and received by all nodes within the collaboration. Hence a unicast of an environment metrics message, E, from one node to the entire collaboration would cost $2(N-1)E$ whereas a multicast of the same information would cost $N \times E$.

Equation 1 models the total network utilization consumed by the existing algorithm of Rossi and Ryan [3] during a complete collaboration session. It is given as the sum of: the total bytes of all metrics propagations; the network utilization of each object due to *distributed* placement; each object migration during every adaptation; and any external network utilizations of the application *(e.g. http request etc.).*

$$\mathrm{Nu(existing)} = 2\mathrm{Np}(\mathrm{E}(\mathrm{N}^2 - \mathrm{N})) + \sum_{i=1}^{O} \mathrm{Nu_{ext}}(\mathrm{O_i}) + \sum_{i=1}^{O}\sum_{j=1}^{m_i}(\mathrm{Nu_j} * \mathrm{NI_j}) + \sum_{i=1}^{Na}\sum_{j=1}^{O^i_{mo}}(\mathrm{SOS_{ij}} + \mathrm{ECS_j}) \tag{1}$$

Np = Number of propagations
E = Serialized size of the environment metrics.
N = Number of nodes within collaboration
O = Number of objects
m_i *= Number of methods of object i*
NI_j*= Number of invocations of method j.*
Nu_j*= Network utilization of method j [6]*
Na= Number of adaptations
SOSij=Serialized volume of object i during adaptation j.
ECS = Executable Codes size
O^i_{mo}*=number of migrated objects during adaptation i*

The network utilization of the *distributed* algorithm is the byte sum of: the adaptation requests sent; the adaptation reports responded for each adaptation; each object's network utilization due to *distributed* placement; each object migration on every adaptation; and any external network utilizations of the application *(e.g. http request etc.)*(NU_{ext}).

$$\mathrm{Nu(distributed)} = \mathrm{Na(nR)} + \sum_{i=1}^{Na} \mathrm{Nf_i}\,(\mathrm{N} \times \mathrm{F}) + \sum_{i=1}^{O} \mathrm{NU_{ext}}(\mathrm{O_i}) + \sum_{i=1}^{O}\sum_{j=1}^{m_i} (\mathrm{Nu_{ij}} * \mathrm{NI}) + \sum_{i=1}^{Na}\sum_{j=1}^{O^i_{mo}} (\mathrm{SOS_{ij}} + \mathrm{ECS_j}) \tag{2}$$

To visualize the network utilization of each algorithm under varying numbers of: metrics propagations (*Np*); adaptations (*Na*); and collaboration sizes (*N*), a constant upper bound for the number of fitness values returned is first assumed (*Nf=N-1).* Setting *Nf* to its upper bound for each adaptation models the scenario in which no functional delay scheme exists in the new algorithm, thereby providing a worst case comparison, in terms of network utilization, of the *distributed* algorithm and the *existing* approach.

Figure 3 & Figure 4 show a 3D region of the values of *Na, Np* and *N* for which the network utilization of the *existing* algorithm would be better than the *distributed* approach and vice-versa. The region is determined based on a simple 3D *regionplot* performed using the computational tool Mathematica [16] for the ty $Nu(existing) < Nu(distributed)$ and vice-versa, under the constraints specified in section 4.1. Note that the costs incurred by external application network utilization, object placement and object migration cancel out as they are assumed to be the same for both algorithms.

Figure 3, shows that the *existing* algorithm provides comparatively better network utilization in small collaboration sizes and very high numbers of adaptations ($Na \geq 65\% Np$). In larger collaboration sizes, the *existing* algorithm provides better performance only at increasingly higher numbers of adaptations where $Na \cong Np$. Computing for the maximum disparity between the two algorithms within this region, shows that the *existing* algorithm could offer as much as 35% less network utilization when NA=NP.

Figure 4, shows the complementary region, under which the *distributed* algorithm provides better resource utilization. The algorithm provides better network utilization under a greater portion of the environment settings. While the algorithm provides better network utilization under relatively less numbers of adaptations, in small collaborations, it is able to outperform the *existing* approach even under high numbers of

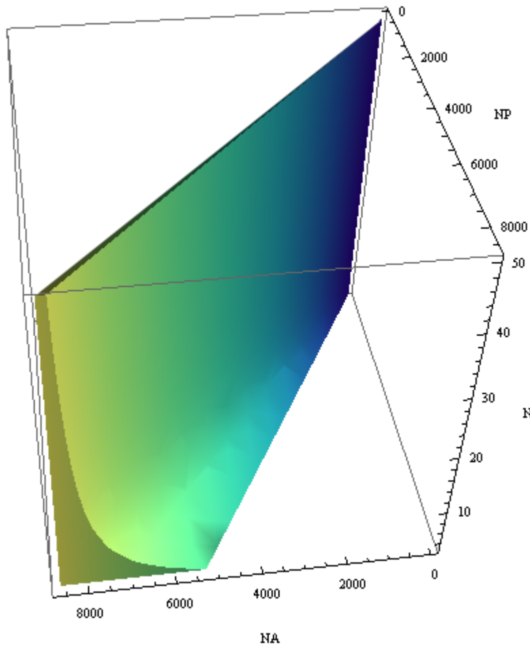

Fig. 3. The region of N, Na and Np for which the *existing* approach provides better network utilization. $\boldsymbol{Nu(existing) < Nu(distributed)}$

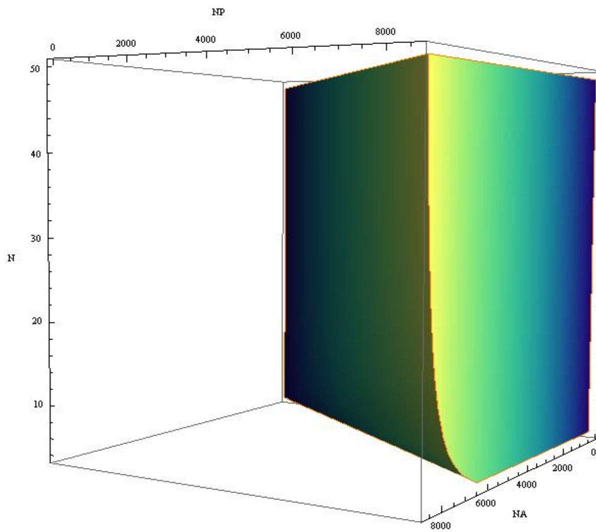

Fig. 4. The region of N, Na and Np for which the *distributed* algorithm provides better network utilization over the *existing* approach. $\boldsymbol{Nu(distributed) < Nu(existing)}$**.**

adaptations in larger collaboration environments. Computing for the maximum degree of network utilization difference between the two algorithms under this region shows that the proposed *distributed* approach can provide over 900% less network utilization under relatively low number of adaptations (Na<10%Np).

In practice, a scenario in which the number of adaptations within the collaboration is as high as that favoring the *existing* algorithm is unlikely even in extremely dynamic computing environments. Such a high adaptation count suggests an object topology in a constant state of flux, thus implying that adaptation decisions are performed without benefits being gained, whilst adaptation overheads are still incurred. Furthermore an event in which a node exceeds its resource utilization threshold, thus requiring an adaptation, would be a far less frequent occurrence than metrics propagation occurring for lesser degrees of change in resources.

The above figures show that even without a delay scheme to reduce the number of fitness reports, the *distributed* algorithm provides better network utilization for a greater range of possible environmental scenarios and is more scalable with regard to the collaboration size and application execution duration.

The effect of a delay scheme on the results discussed above is shown in Table 2, wherein it is evident that more effective delay schemes reduce the maximum degree with which the *existing* algorithm outperforms the *distributed* solution until a delay period which is 55% effective (i.e. only 45% of the collaborating nodes reply on each adaptation) results in the *distributed* algorithm always outperforming the *existing* one. This can also be visualized as the shrinking of the region identified by Figure 3 and the expansion of the region shown in Figure 4.

Table 2. The effect of a delay scheme on the network utilization disparity between the algorithms

Delay Scheme % effectiveness	Maximum Degree with which *existing* can outperform *distributed*
0%	2.16 GB
5%	181.44 MB
25%	43.2 MB
50%	25.9 MB
55%	-19.44KB
75%	-54.90 KB
95%	-2.28 MB

4.3 Memory Utilization

Memory consumption is defined as the number of bytes stored by each algorithm for the duration of the collaboration. This excludes temporary memory resident data such as adaptation request information from other nodes, or storage of serialized environmental metrics before propagation. The assumption also disregards memory utilization of the middleware as it would not bear upon the comparison of the individual algorithms.

The global memory consumed in the collaboration by the *existing* algorithm of Rossi and Ryan [3] shown in Equation 3, is the sum of the memory consumed by: environment metrics of the entire collaboration stored on every node; the metrics information stored about each object; memory utilization of each object; and the memory utilization of the middleware framework.

Similarly, the total memory consumed by the *distributed* algorithm, shown in Equation 4, is the sum of: the environmental metrics of each node; the metrics information stored about each object; memory utilization of each object; and memory utilization of the middleware framework.

$$\mathrm{Mu(existing)} = \mathrm{N^2E} + \sum_{\mathrm{i=1}}^{\mathrm{O}} \mathrm{Mu(O_i)} + \sum_{\mathrm{i=1}}^{\mathrm{O}} \mathrm{Mu(B_i)} \tag{3}$$

$$\mathrm{Mu(distributed)} = \mathrm{NE} + \sum_{\mathrm{i=1}}^{\mathrm{O}} \mathrm{Mu(O_i)} + \sum_{\mathrm{i=1}}^{\mathrm{O}} \mathrm{Mu(B_i)} \tag{4}$$

Equation 3 models the memory utilization of the *existing* approach. Equation 4 models the memory utilization of the *distributed* algorithm.

O=Number of mobile objects
E = Environmental metrics size
N = number of nodes
Mu(x) = Average Memory utilization of x
B_i =Object metrics of object i

Equation 3 and 4 show that while the *existing* algorithm has a global memory complexity of $O(N^2)$ for storing collaboration information on every node, the *distributed* approach has a more favorable memory utilization complexity of *O(N)*. Figure 5 shows that the difference between the memory utilization of both algorithms increases exponentially with the increase in number of nodes and linearly with the increase in environmental metrics size. The *distributed* algorithm hence provides increasingly better memory utilization with an increase in the heterogeneity and size of the collaboration.

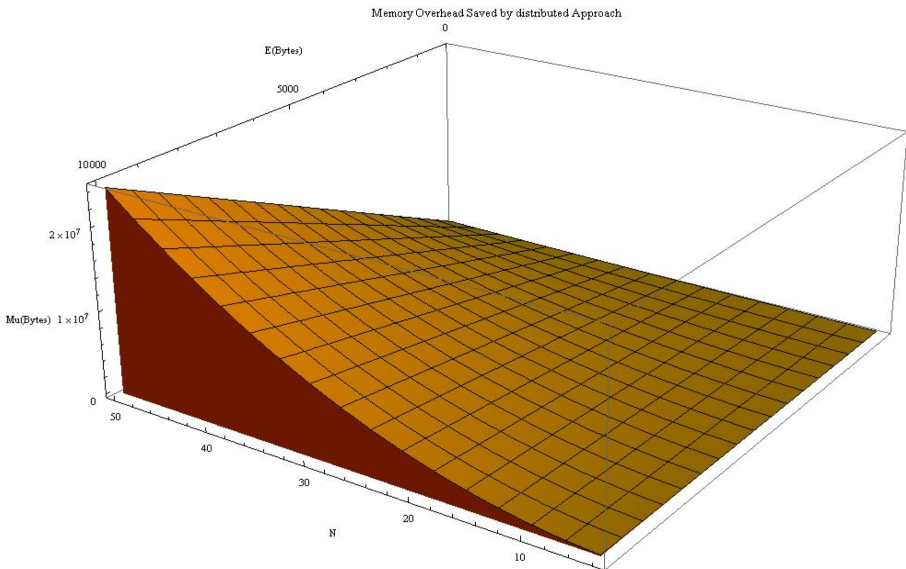

Fig. 5. Memory overhead saved by the *distributed* algorithm compared to the *existing* algorithm

5 Summary and Conclusion

The paper has proposed a *distributed* approach to local adaptation decision making to improve the *efficiency* and *scalability* of *existing* approaches. The network and memory utilization of the approach was mathematically modeled and compared to an *existing* algorithm. The results showed the *distributed* approach to be more scalable with regards to collaboration size and diversity of the computation environment, as well as offering over 900% better network utilization in large scale dynamic collaborations while still maintaining a linear memory utilization complexity.

Future work will focus on implementing and empirically evaluating the *effectiveness* of the algorithm's adaptation outcomes for the objective of application performance improvement. The use of a clustering technique, and an effective delay period will also be investigated and their performance impact analyzed.

References

1. Kim, M., Copeland, J.A.: Bandwidth sensitive caching for video streaming application. In: IEEE International Conference on Communications, 2003. ICC 2003 (2003)
2. Hütter, C., Moschny, T.: Runtime Locality Optimizations of Distributed Java Applications. IEEE Computer Society Press, Washington (2008)
3. Rossi, P., Ryan, C.: An Empirical Evaluation of Dynamic Local Adaptation for Distributed Mobile Applications. In: Proc. of 2005 International Symposium on Distributed Objects and Applications (DOA 2005), Larnaca, Cyprus (2005)
4. Ryan, C., Westhorpe, C.: Application Adaptation through Transparent and Portable Object Mobility in Java. In: Proc. of 2004 International Symposium on Distributed Objects and Applications (DOA 2004), Larnaca, Cyprus (2004)
5. Felea, V., Toursel, B.: Adaptive Distributed Execution of Java Applications. In: 12th Euromicro Conference on Parallel, Distributed and Network-Based Processing, PDP 2004 (2004)
6. Gu, X., et al.: Adaptive Offloading for Pervasive Computing. IEEE Pervasive Computing 3(3), 66–73 (2004)
7. Tilevich, E., Smaragdakis, Y.: J-orchestra: Automatic java application partitioning. In: Magnusson, B. (ed.) ECOOP 2002. LNCS, vol. 2374, pp. 178–204. Springer, Heidelberg (2002)
8. Philippsen, M., Zenger, M.: JavaParty Transparent remote objects in Java. In: Proc. ACM 1997 PPoPP Workshop on Java for Science and Engineering Computation (1997)
9. Fahringer, T.: JavaSymphony: A System for Development of Locality-Oriented Distributed and Parallel Java Applications. In: Cluster 2000. IEEE Computer Society Press, Los Alamitos (2000)
10. Garti, D., et al.: Object Mobility for Performance Improvements of Parallel Java Applications. Journal of Parallel and Distributed Computing 60(10), 1311–1324 (2000)
11. Sakamoto, K., Yoshida, M.: Design and Evaluation of Large Scale Loosely Coupled Cluster-based Distributed Systems. In: Li, K., Jesshope, C., Jin, H., Gaudiot, J.-L. (eds.) NPC 2007. LNCS, vol. 4672, pp. 572–577. Springer, Heidelberg (2007)
12. Ou, S., Yang, K., Liotta, A.: An adaptive multi-constraint partitioning algorithm for offloading in pervasive systems. In: Fourth Annual IEEE International Conference on Pervasive Computing and Communications, 2006. PerCom 2006 (2006)

13. Ryan, C., Rossi, P.: Software, performance and resource utilisation metrics for context-aware mobile applications. In: 11th IEEE International Symposium in Software Metrics (2005)
14. Gani, H., Ryan, C., Rossi, P.: Runtime Metrics Collection for Middleware Supported Adaptation of Mobile Applications. In: International Workshop on Adaptive and Reflective Middleware, ACM Middleware, 2006, Melbourne, Australia (2006)
15. Abebe, E., Ryan, C.: Online Appendix: Decision Computation Calculations for a Distributed Approach to Local Adaptation (2009), http://goanna.cs.rmit.edu.au/~eabebe/DOA2009/Abebe_Ryan_Appendix.pdf
16. Wolfram Research, Wolfram Mathematica 7 (2009)