The Knowledge Engineering Review, Vol. 21:3, 205–219. © 2006, Cambridge University Press
doi:10.1017/S0269888906000890 Printed in the United Kingdom

Policy-based techniques for self-managing parallel applications

RICHARD JOHN ANTHONY

Department of Computer Science, University of Greenwich, London SE10 9LS, UK;
e-mail: r.j.anthony@gre.ac.uk

Abstract

This paper presents an empirical investigation of policy-based self-management techniques for parallel applications executing in loosely-coupled environments. The dynamic and heterogeneous nature of these environments is discussed and the special considerations for parallel applications are identified. An adaptive strategy for the run-time deployment of tasks of parallel applications is presented. The strategy is based on embedding numerous policies which are informed by contextual and environmental inputs. The policies govern various aspects of behaviour, enhancing flexibility so that the goals of efficiency and performance are achieved despite high levels of environmental variability. A prototype self-managing parallel application is used as a vehicle to explore the feasibility and benefits of the strategy. In particular, several aspects of stability are investigated. The implementation and behaviour of three policies are discussed and sample results examined.

1 Introduction

This paper presents an investigation into ways that policy-based autonomics can be integrated into parallel applications that execute in loosely-coupled systems, and the extent of the benefits that can be achieved. The adopted implementation strategy involves embedding self-configuring and self-optimizing capabilities (Koehler *et al.*, 2003; Kephart & Chess, 2003) into applications to enable continuous adaptation to the context and run-time environment to maintain performance and efficiency.

A complexity problem arises in the configuration and management of systems of interacting components, especially as the scale of the systems, and the sophistication of the components, increases. The field of autonomics currently enjoys rapidly growing interest; fundamentally because it offers a solution to this problem. The autonomic computing paradigm advocates self-managing behaviour in which applications modify their behaviour to suit their environment and context; see, for example, Badr *et al.* (2004). This approach reduces the emphasis on the preconfiguration of components, and relies instead on inbuilt learning and discovery capabilities.

The IBM autonomics model (IBM, 2004) provides a popular framework in which self-management is divided into four specific self-categories (configuration, healing, optimization and protection). Using this model as a reference point, this paper is concerned with self-configuration and self-optimization. Additionally, this paper is particularly concerned with self-stabilization, which is a concept that can be considered to cut across all of the other self-behaviours. A system or application needs to be able to adapt to changes in its environment and operating context whilst remaining stable. The exact fine-grained behaviour at any moment may not be precisely knowable to an outside observer but it is fundamentally important that the more general, high-level behaviour be predictable and 'legal' as defined by the semantics of the application itself.

https://doi.org/10.1017/S0269888906000890 Published online by Cambridge University Press

Policy-based computing is a versatile and accessible means of implementing autonomics, because it is relatively easy to codify a wide range of behavioural parameters into policy rules. In particular, this is also a viable approach for retrofitting autonomic behaviour into legacy applications, as the policy can be kept separate from the mechanism, so that modifications need only be made at a few carefully selected interface points.

As LAN technology continues to improve in terms of reliability and performance, loosely-coupled systems are increasingly popular platforms for parallel processing. Cluster and Grid configurations provide a viable, relatively cheap and commonly available alternative to specialised tightly-coupled systems (Hobbs & Goscinski, 2000) for a wide variety of coarser-grained parallel applications. Often, the physical resources that constitute the cluster or Grid are already installed and thus represent a free resource. For example, it is common to use 'standard' office PCs interconnected with LAN technology, reconfigured with a Grid or cluster software layer for out-of-hours processing of computationally-intense applications when the physical resource would otherwise be idle.

However, loosely-coupled environments can be highly dynamic (Shum, 1999; Weng *et al*., 2004). Congestion is dynamic in the time, location, and severity of its formation and causes proportionally dynamic communication delay. Processing nodes are rarely performance-homogenous, differing in many ways, including: memory capacity, processing speed and secondary storage capacity. The number of processing nodes available can also vary over time. A workstation cluster or Grid is not always dedicated to the execution of a single application. The presence of other tasks constitutes a background workload with which a deployed task must compete for resources. Therefore, the effective processing capacity of nodes (as seen by a specific task) is variable. The extent of the dynamism varies from system to system and over time within systems. The longer the run-time of a parallel application, the more exposed it is to changes in its environment that effect its execution efficiency. Thus, static deployment in loosely-coupled environments is potentially highly inefficient.

2 Parallel processing in loosely-coupled systems

Much theoretical work has been done to solve aspects of the dynamic decomposition and dynamic deployment problems. For example, Jin & Liu (2004) discuss an agent-based solution, but assumes that Grid nodes are homogenous and that applications are arbitrarily divisible. Such approaches make the fundamental assumption that loosely-coupled systems are high-communication-latency versions of tightly-coupled systems; for example, Goscinksi (1999) advocates that *homogenous* clusters of workstations (COWs) should be used for parallel processing, to avoid problems associated with computational imbalance.

Dynamic decomposition remains a tricky problem (Hendrickson, 2000) and thus, commonly, static task decomposition and deployment techniques are used when adapting parallel applications for execution in loosely-coupled environments such as clusters and Grids; see, for example, Gottlieb (2001). Generally, the partitioning is related to a pre-determined deployment plan; for example, the run-time system is known to contain a certain number of machines with (assumed) similar processing capacities and connectivity, therefore the partitioning is carried out appropriately (subject to limitations of the application architecture); see, for example, Allen (2005), Grandison *et al.* (2005) and Du *et al*. (2005).

However, loosely-coupled systems typically comprise resource-heterogeneous nodes and are much more dynamic in nature than tightly-coupled systems. The sources of dynamic behaviour include:

1. asynchronous scheduling and communication arising from the autonomy of the individual nodes;
2. background workload variability, which influences the effective performance of nodes;
3. independent failure and recovery of worker nodes, and the arrival of new workers;
4. end-to-end communication delay, impacted by network congestion which varies between locations and over time.

https://doi.org/10.1017/S0269888906000890 Published online by Cambridge University Press

The suitability of a particular application for parallel deployment over a loosely-coupled system is determined fundamentally by the granularity of tasks. When executing in a loosely-coupled environment the granularity (expressed as the ratio of computation time to communication time) is variable because both factors are subject to variability in the environment, as discussed above. The granularity is acceptably coarse only when the computational time to communication time ratio is sufficiently high that the worker is used efficiently and contributes to an overall speedup. This must be determined dynamically for each worker node. The dynamic behaviours of loosely-coupled environments are a barrier to achieving optimal or efficient performance if static application configurations are used, and can be problematic for externally-managed run-time configuration (Ibrahim *et al*., 2004; Barrett *et al*., 2004).

The proposed strategy is to embed self-managing capabilities (in the form of numerous adaptive policies) into the components of parallel applications so that the run-time behaviour, including deployment, can be continuously fine-tuned during execution to maintain high performance and efficiency despite environmental fluctuations. The overall goal is to minimise the makespan by using the available resources efficiently.

Dynamic configuration could be introduced either at the point of decomposition into tasks, or during the deployment of tasks over processing nodes. To avoid most of the complexity, and still achieve most of the potential gains (and the advantage of generalization), it was decided to adapt the *deployment* dynamically. The decomposition is done prior to run-time because, generally, this is governed by problem-specific constraints (see, for example, Hu & Zou (2005)), and also because dynamic decomposition is generally a hard problem, although Clementi & Corongiu (1999) describe a dynamic decomposition that is performed at the level of subroutines. The strategy requires that all run-time influence is removed from the decomposition (partitioning and deployment should be regarded as separate unrelated activities). To provide the maximum deployment flexibility, the partitioning should be carried out at the finest feasible granularity, given the internal architecture of the application code (i.e. without regard to the characteristics of the run-time environment). The actual deployment is dynamic; tasks are created by the client component during application execution. The number and size of tasks is continually tuned to suit the available worker population, taking into account their individual performance, workloads and connectivity. Run-time flexibility is enhanced by dynamic discovery and incorporation of workers.

Figure 1 contrasts static decomposition implicitly determined by a deployment plan (i.e. the run-time system is known to comprise three nodes, so the application is coarsely decomposed accordingly), with finer-grained decomposition that is completely free of any run-time influence. In the example shown, three worker nodes are available at initialization; a fourth becoming available mid-execution. The figure illustrates the deployment flexibility benefit of finer-grained decomposition; the allocation can be adjusted dynamically to reflect changes in the effective performance of worker nodes and the available worker population.

3 Policy-based autonomics

Policy-based systems use one or more policies dynamically to inform their run-time behaviour; examples include Ananthanarayanan *et al.* (2005) and Maglio *et al.* (2005). A policy can be selected from a library of policies, and initialised from a template of preferences. Policies can be subsequently adapted to suit environmental or application-specific conditions.

Policy-based autonomic systems require certain capabilities:

- monitor the current system;
- identify problems, inefficiencies or anomalies (generally, where the system state diverges away from its goal state);
- devise possible solutions (to move the system to within an acceptable distance from the goal state);
- apply the solutions (in such a way that erratic sudden changes are avoided to maintain stability).

https://doi.org/10.1017/S0269888906000890 Published online by Cambridge University Press

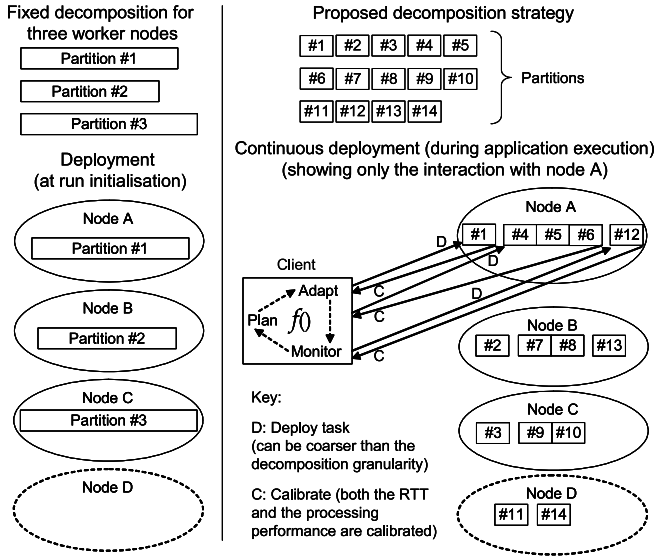


Figure 1 Contrasting diverse approaches to decomposition and deployment

These activities form a loop in which the system continually modifies its behaviour to match the demands of its environment. Certain conditions may trigger the replacement of the current policy with another from a library, or modification of the current policy rules. In this way the system's behaviour is not only responsive to its environment; it can also evolve as trends are identified. In the specific context of a policy governing parallel application deployment, the monitor aspect is concerned with activities such as the calibration of the effective processing speed at each node and the communication cost of communicating with that node. The planning aspect is concerned with determining whether the current configuration is appropriate for the context and environment, or whether any configuration changes can be applied better to optimize the deployment. The adaptation aspect is concerned with managing the deployment over the longer term by updating the policy configuration.

3.1 Defining and achieving stability in policy-based systems

Autonomic systems must reach a balance between freedom to adapt to ambient conditions, and stable behaviour which is 'legal' as defined by the application's semantics. One feasible solution is to identify 'bounds' to constrain the operational envelope of components (Ibrahim *et al.*, 2004).

Figure 2 provides a simplified illustration of bounded behaviour in which the policy has only one dimension of freedom. The upper and lower bounds can be set externally and passed into the policy at initiation. So long as the policy remains with these bounds, its behaviour is considered legal. However, there is no requirement that the bounds are fixed (as depicted in the left-hand side of the figure); they may be linked to another dynamic aspect of the system (as depicted in the right-hand side of the figure). Clearly the policy can be more or less restricted by controlling the margin between the bounds. It is important to realise that, in such a system, even when it exhibits 'legal' behaviour it is not possible to state *exactly* what it will do next — if we wish to be that prescriptive we are defeating the object of self-adaptability.

There will always be some environmental randomness and hysteresis in the control and feedback mechanisms, so it is not possible to perfectly track a goal state that is itself dynamic. Attempting a perfect track can lead to oscillation arising from the continual fine adjustments, leading to additional corrective work. However, a dead zone can be introduced either side of the goal state such that, once inside this zone, the policy makes no adjustment.

https://doi.org/10.1017/S0269888906000890 Published online by Cambridge University Press

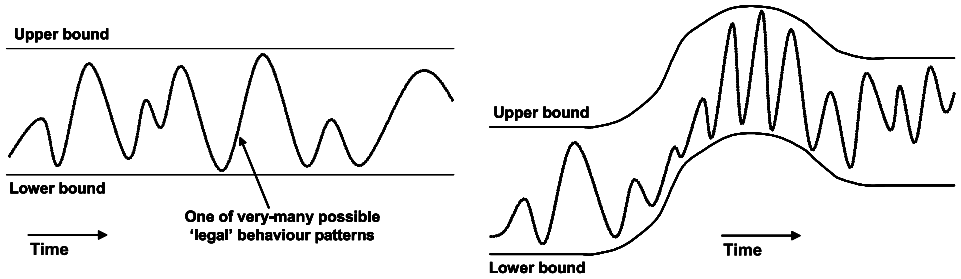


Figure 2 Policy behaviour scoped by externally defined bounds

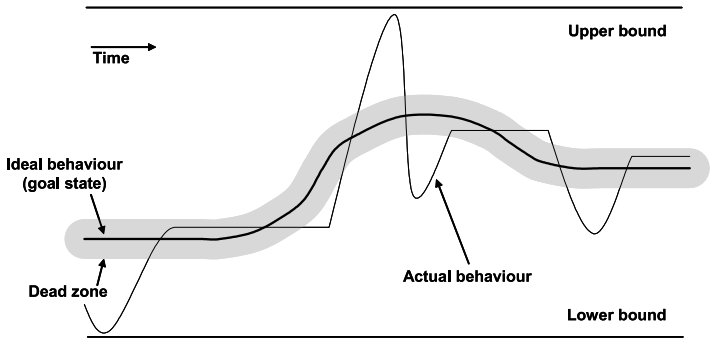


Figure 3 Tracking goal state behaviour using a ''dead zone''

Figure 3 illustrates how the dead zone enhances stability by reducing the amount of behavioural 'noise', especially micro-oscillation around the goal state. If the goal state changes significantly, the policy eventually finds itself outside the dead zone and once again must adapt the system.

As illustrated in Figures 2 and 3, it is very likely that the policy is tracking a goal state that is itself dynamic, being influenced by environmental conditions. Because of this, it is important to consider the rate of adaptation (in terms of the amount of adjustment applied at any single step). The significance of this issue is relative to the extent of dynamism in the end goal. An adaptation strategy is defined, in which stability is ranked as the highest priority of the non-functional requirements for autonomic systems (followed by performance, scalability, etc.). Purposely, to avoid erratic behaviour and oscillation, successful adaptation is defined as 'moving the system from its current state *towards* the *currently perceived* optimum state'. In this way a gradual convergence is achieved. To implement the strategy, the concept of an adaptation factor A is introduced which governs the fraction of the calculated adjustment (based on current distance from the perceived optimum state at that moment) that is actually applied. The adaptation is most aggressive when A = 1.0 (which means that the policy output at time t+1 is the value of the system goal at time t) but can be locally unstable. Conversely, convergence might not occur, or may occur too slowly, if a low value (such as $A = 0.25$) is used.

Figure 4 indicates how lower values of A lead to more-gradual, but also less-erratic tracking of the perceived 'ideal' behaviour. In the first graph the system goal follows a steady upward trend; in the second graph the system goal follows an upward trend but is locally unstable; in the third graph the system goal is unsteady.

A clear example of locally-bounded behaviour is provided by the round-trip-time (RTT)-measurement-optimization policy. The dead zone and adaptation factor concepts are exemplified by the granularity policy. Section 5 discusses the implementation details for these policies.

It is worth noting (although outside of the scope of this paper), that all of the parameters that might be short-term fixed with respect to a specific low-level policy, such as the bounds, the size of the dead zone and the value of the adaptation factor, could all be determined over a longer time frame by higher-level policies, arranged hierarchically.

https://doi.org/10.1017/S0269888906000890 Published online by Cambridge University Press

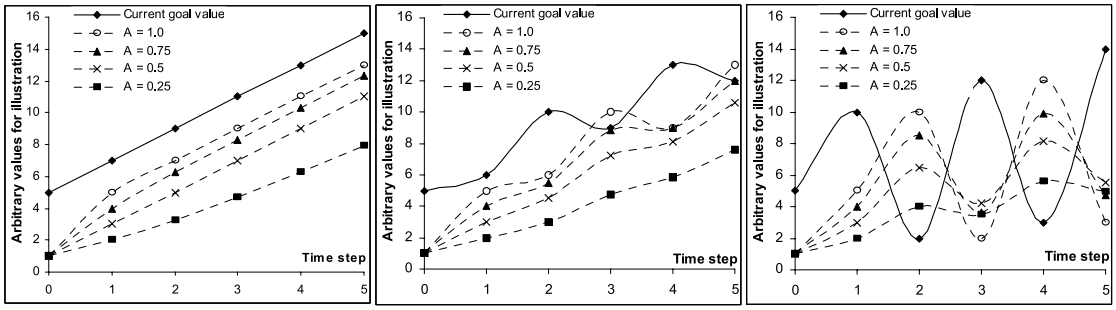


Figure 4 The effect of different values of the adaptation factor (A) with different system behaviours

4 A self-managing parallel application

To demonstrate the feasibility of the strategy, a self-managing parallel application has been developed. Several embedded policies govern the way in which the application behaves in a given context. This context arises from a combination of application features and behaviour, and environmental conditions. A 'client' dynamically discovers workers during execution and adjusts its deployment function to issue a share of the work accordingly. Similarly, workers who disappear are factored out and the allocations to remaining workers adjusted.

In addition to the processing done at worker nodes, the client is able to execute a worker thread locally in certain execution contexts. This can shorten the total run-time, and is useful to ensure eventual completion when no workers are available. As the number of remote workers increases, more effort is required to manage them, and the benefit of local processing at the client is diminished in terms of the effect on makespan. Thus, a threshold is crossed at which the client ceases to execute the local worker thread. The value of this threshold can be adjusted to reflect the relative processing performance of the client and worker nodes.

The demonstration implementation internally uses a very fine-grained calculation, Monte-Carlo approximation of Pi, in which there are no causal relationships nor communication between the individual calculations. This approach facilitates exploration of a wide range of dynamic deployment behaviours by permitting many different 'applications' to be emulated. This emulation is achieved by means of a user-supplied minimum granularity value g which limits the minimum task size to an integer number of the internal calculations. Thus, g defines an 'application' in terms of its *decomposition* granularity. For example, if g = 1 million, then the client must issue task sizes to workers of integer multiples of 1 million of the Monte-Carlo computations, i.e. 1 million, 2 million, etc. (this is the dynamically-determined *deployment* granularity). No upper limit is imposed on the deployment granularity. The granularity policy has rules to ensure that work can be distributed efficiently and thus utilize workers as flexibly as possible.

If the run-time system represents a homogenous and highly stable environment it would be efficient simply to divide up the work in accordance with the instantaneous number of workers existing and issue the coarsest-grained tasks algorithmically possible. Such an approach becomes less attractive in more dynamic environments. For example, initial allocation of the entire workload would be counterproductive if an additional worker subsequently became available during a long-running execution and could not be utilized, or a long-running worker crashed just before reporting its results. Thus, in dynamic environments the client should allocate smaller tasks to retain scheduling flexibility.

In order that the deployment can be maintained within an efficient operational envelope, the processing performance of worker nodes, and the RTT between the client and each worker is measured periodically. This enables the work to be distributed in proportion to nodes' instantaneous processing capacities, obviates the need to relocate work for the achievement of load balance, and is more transparent with respect to node failure because a smaller fraction of the total work is lost (and is subsequently reissued to another worker).

https://doi.org/10.1017/S0269888906000890 Published online by Cambridge University Press

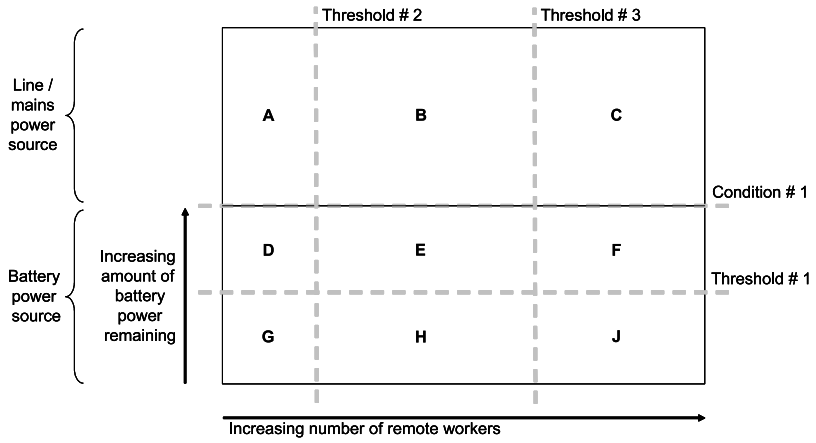


Figure 5 The local-worker-policy behaviour-space is divided into regions by the policy parameters

Condition #1	`ALLOW_LOCAL_WORKER_WHILE_ON_BATTERY_POWER = true`
Threshold #1	`LOW_BATTERY_POWER_THRESHOLD_PERCENT = 50`
Threshold #2	`MINIMUM_ACCEPTABLE_NUMBER_OF_WORKERS = 0`
Threshold #3	`MAX_WORKERS_WHILE_RUNNING_LOCAL_WORKER = 7`

Figure 6 Template values for the local-worker-thread policy

5 Selected policies

Several policies have been implemented to facilitate the investigation and evaluation of autonomic behaviour in the context of parallel processing. Each policy controls a specific aspect of behaviour (although the policies interact indirectly; for example, one policy may govern the way in which measurement data is collected whilst another policy may use this measurement data to adjust workload deployment). This paper concentrates on three of the policies, which are described in turn below.

The local-worker-thread policy determines whether the client should execute a local worker thread or should rely entirely on the remote workers to complete the computation. This policy takes into account contextual conditions (such as the instantaneous number of remote worker nodes), and environmental conditions (such as whether the client's power source is mains or battery, and how much battery life remains). The policy ensures that the local worker thread is executed only when it is both affordable and beneficial. The relative value of the local worker thread diminishes as the number of remote workers increases (in terms of its effect on the overall application makespan, and because of the increased overhead of managing the remote workers).

The policy uses three threshold values and one boolean condition to divide up the possible behaviour space into distinct regions, as illustrated in Figure 5. These are set, initially, by means of a template of preference values. By keeping the policy logic separate from the implementation mechanism, it is possible for these values to be subsequently adjusted by the policy itself or by another policy operating at a higher level.

Figure 6 shows typical template values for the local-worker-thread policy and shows how the condition and thresholds map onto the behaviour-space graph (Figure 5).

The local-worker-thread policy logic, shown in Figure 7, involves interactions between contextual information from the policy variables (which are shown italicized) and environmental information such as the remaining battery life and the instantaneous number of remote workers present (which are shown underlined). The resulting behaviour directives are shown in bold. The policy logic has been annotated to provide a mapping of the policy behaviour onto the relevant regions of the behaviour-space graph (Figure 5).

The RTT-measurement-optimization policy governs the measurement of the RTT between the client node and each worker. Performance measurement information (which includes the RTT as well as the effective processing performance of each node), forms part of the execution context and

https://doi.org/10.1017/S0269888906000890 Published online by Cambridge University Press

```
if(PowerStatus.ACLineStatus == 1) // Mains power (regions A, B, C)
{
   if(iNumberOfWorkersConnected > iMaxWorkersWhileRunningLocalWorker) // Threshold #3
   {
       return NO_TooManyRemoteWorkers; // Region C
   }
   else
   {
       return YES_MainsPower; // Regions A and B
   }
}

else  // Battery Power (regions D, E, F, G, H and J)
{
   if(true == bAllowLocalWorkerWhileOnBatteryPower) // Condition #1
   {
       if(PowerStatus.BatteryLifePercent > iLowBatteryPowerThresholdPercent) //Threshold #1
       {
           if(iNumberOfWorkersConnected > iMinimumAcceptableNumberWorkers) // Threshold #2
           {
               return NO_BatteryPowerAndRemoteWorkerAvailable; // Regions E and F
           }
           else
           {
               return YES_BatteryPowerButNoRemoteWorkers; // Region D
           }
       }
       else
       {
           return NO_BatteryLowWaitForRemoteWorker; // Regions G, H and J
       }
   }
   else
   {
       return NO_NotAllowedOnBatteryPower; // Regions D, E, F, G, H and J
   }
}
```

Figure 7 The local worker-thread self-configuration logic

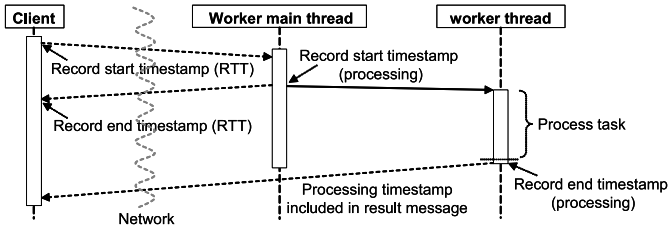


Figure 8 Calibration of worker performance and RTT

is used to determine the efficiency of using each node as a worker and the appropriate task size to issue to that node. Measurements are performed using live problem data so that they form part of the useful work done and are not entirely overhead. This approach also ensures that the calibration is reliable, because the resource-usage signature on the node is true to the real problem. The amount of time taken to process a specific-sized task is used to determine the current processing performance of each worker, taking into account its ambient workload. The RTT, which includes the round-trip communication latency as well as the scheduling delay at the worker in responding to the message, is used by the granularity policy to determine the relative speedup achieved by each worker which in turn is used to identify inefficient/ineffective workers that should be rejected. Figure 8 depicts the worker thread calibration.

The frequency of RTT measurement can be optimized to suit the variability of the network behaviour. The more stable the network, the less frequent the RTT is measured, saving some processing overheads and messages. The policy is differently parameterised for each worker, based on their current and previous behaviour and environmental influences. RTT is measured the first time a task is issued to a particular worker. For subsequent tasks, if the current RTT measurement is within an acceptable distance from the previous value (the dead zone is determined to be, by default, ± 10% of the previous value), a count of similar RTT values is incremented, otherwise it is reset to zero. When a sufficient number of similar values occur in sequence, the policy allows a

https://doi.org/10.1017/S0269888906000890 Published online by Cambridge University Press

```
ALLOW_REMOTE_WORKER_RTT_CALIBRATE_OPTIMIZATION = true
RTT_CALIBRATE_INTERVAL_MAX = 5                    // Calibration Interval Upper bound
RTT_CALIBRATE_INTERVAL_INITIAL = 1                // Calibration Interval Lower bound
RTT_CALIBRATE_DIFFERENCE_THRESHOLD_PERCENT = 10
RTT_CALIBRATE_SAME_COUNT = 3
```

Figure 9 Worker-RTT calibration policy template values

number of RTT calibrations to be skipped (the 'calibration interval'). This saves some computation and, more importantly, some messages and thus outweighs the costs of the policy itself. The calibration interval is dynamically incrementally adjusted up to a policy maximum (the upper bound) if the network remains stable over a longer period. When the period of stability comes to an end, the calibration interval is reset to its lower bound value. The template settings for the policy are shown in Figure 9. Figure 10 illustrates the policy logic, showing the interaction between the context variables (underlined) and policy variables (italicized) in determining whether the RTT should be calibrated the next time the worker is sent a task. The resulting behaviour directives are shown bold.

Figure 11 provides an example of the behaviour of the policy when operating in a non-dedicated wired-LAN in which significant load fluctuations occur. For maximum clarity, the graph shows data pertaining to only a single worker. When there is low variance in the RTT over successive measurements the policy increments the calibration interval (the number of consecutive RTT messages that can be skipped) until it hits the upper bound (which was set at 5 in the template values). Small changes in the RTT value are ignored because a relative tolerance zone (dead zone) of ± 10% of the previous RTT value is used. When the RTT changes sufficiently sharply to move outside of the tolerance zone the policy adjusts the calibration interval back to the lower bound (which was set at 1 in the template values). The graph shows a trace of 352 work grains issued to the remote worker, and in 184 cases the RTT calibration was skipped, representing a saving of 52.27% of calibration messages. The policy strives to maintain an optimal balance between saving messages as the network load increases, and performing sufficient measurement so that the application deployment can be appropriately tuned.

The granularity policy is responsible for dynamically maintaining the deployment of work over the available workers such that each task issued to a specific worker reflects its current effective processing capacity. In addition, the granularity policy must incorporate newly-arriving workers; reallocate work when existing workers depart (which is taken to include node crashes and mobile nodes moving out of contact); and reject workers that are determined to be inefficient.

In order dynamically to achieve a balance of the workload over the available workforce, and to be able to incorporate new workers mid-execution, the policy must carefully determine (for each worker) the appropriate size of tasks (the deployment granularity). It is undesirable to issue too-large tasks for several reasons.

1. It may prevent the inclusion of a subsequently-discovered worker (if all the work has been allocated).
2. It requires expensive reallocation if conditions change (this is often not facilitated, but see Baiardi *et al.* (2001), for an example).
3. Worker failure represents a significant cost in terms of wasted processing effort and the delay incurred while the processing is re-done. Reducing the deployment granularity increases the retained scheduling flexibility, allowing the policy to be responsive to changes in the number of workers and their effective processing capacities.

However, it is undesirable to issue too-small tasks because of the additional communication overheads and management overheads that are incurred. The decomposition granularity (g) provides a lower limit on the deployment granularity.

An application-wide flexibility factor (F) is introduced to facilitate dynamic calculation of the initial deployment granularity for workers based on the size of, and extent of variance in, the worker population. The initial task size is given by W_R / FN where W_R is the instantaneously

https://doi.org/10.1017/S0269888906000890 Published online by Cambridge University Press

```
if(-1 == RttPolicy.iSameCount)  // Initialization (-1 signifies that the relevant worker has not been calibrated previously)
{
   RttPolicy.iSameCount = 0;
   RttPolicy.iPreviousRTT = 0;
   RttPolicy.iCurrentRTT = 100;
   RttPolicy.bCanSkipCalibration = false;
   RttPolicy.iRttCalibrateIntervalCurrent = iRttCalibrateIntervalInitial;
   RttPolicy.iNoCalibrateCount = 0
   RttPolicy.bCalibrateBecauseIntervalReached = false;
   return YES_FirstTime;
}
if(true == RttPolicy.bCanSkipCalibration)  // Stable, skip RTT measurement if the Calibrate Interval has not been exceeded
{
   if(RttPolicy.iNoCalibrateCount < RttPolicy.iRttCalibrateIntervalCurrent)  // OK to skip a measurement
   {
      RttPolicy.iNoCalibrateCount++;
      RttPolicy.iPreviousRTT = RttPolicy.iCurrentRTT;
      return NO_Stable;
   }
   else     // Calibrate Interval has been exceeded, must perform an 'Interval' measurement
   {
      RttPolicy.iNoCalibrateCount = 0;
      RttPolicy.bCanSkipCalibration = false;
      RttPolicy.bCalibrateBecauseIntervalReached = true;  // This flag remembers reason for most-recent measurement
      RttPolicy.iPreviousRTT = RttPolicy.iCurrentRTT;
      return YES_Interval;
   }
}

// Determine if in Dead Zone. (The calculation of iCurrentRTT_MIN and iCurrentRTT
// MAX have been omitted for brevity. These values delimit the Dead Zone).
if(RttPolicy.iCurrentRTT >= iCurrentRTT_MIN && RttPolicy.iCurrentRTT <= iCurrentRTT_MAX)
{   // In the Dead Zone, RTT is stable
   RttPolicy.iSameCount++;  // Do not return at this point
{
else   // Outside of the Dead Zone, RTT has changed significantly
{
   RttPolicy.iSameCount = 0;
   RttPolicy.bCanSkipCalibration = false;
   RttPolicy.iRttCalibrateIntervalCurrent = iRttCalibrateIntervalInitial;
   RttPolicy.iNoCalibrateCount = 0;
   RttPolicy.bCalibrateBecauseIntervalReached = false;
   RttPolicy.iPreviousRTT = RttPolicy.iCurrentRTT;
   return YES_Volatile;
}

// Can only drop through to here if the most-recent RTT value was in the Dead Zone (i.e. stable)
if(true == RttPolicy.bCalibrateBecauseIntervalReached)  // Most-recent measurement was Interval
{
   if(RttPolicy.iRttCalibrateIntervalCurrent < iRttCalibrateIntervalMax)
   {
      RttPolicy.iRttCalibrateIntervalCurrent++;     // Still stable after Interval measurement, so increment soft limit
   }                                                // The soft limit must not exceed the upper bound
   RttPolicy.bCanSkipCalibration = true;
   RttPolicy.iNoCalibrateCount = 0;
   RttPolicy.iPreviousRTT = RttPolicy.iCurrentRTT;
   return NO_Stable;
}
if(RttPolicy.iSameCount >= iRttCalibrateSameCount)  // Determine if sufficient stability to start skipping measurements
{
   RttPolicy.bCanSkipCalibration = true;
   RttPolicy.iNoCalibrateCount = 0;
   RttPolicy.iPreviousRTT = RttPolicy.iCurrentRTT;
   return NO_Stable;
}
else
{
   RttPolicy.iPreviousRTT = RttPolicy.iCurrentRTT;
   return YES_Volatile;
}
```

Figure 10 Part of the RTT calibration self-optimization logic

determined amount of work remaining (not allocated), and N is the instantaneously determined number of worker nodes present. The same formula is applied to workers present at the start of processing and to workers that join mid-execution. Optimization of the initial deployment granularity is achieved by dynamically adjusting the value of F, which has a default initial value of 5, to suit the extent of dynamism in the environment. F is incremented each time a remote worker joins or leaves; and is then gradually reduced back to its default value at a slow rate. The effect of

https://doi.org/10.1017/S0269888906000890 Published online by Cambridge University Press

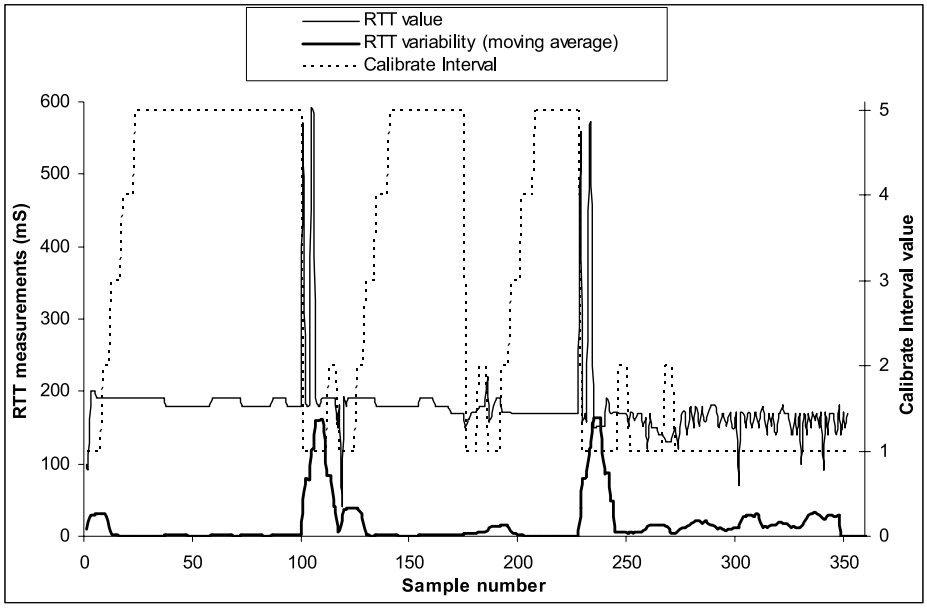


Figure 11 A sample of the behaviour of the RTT-measurement-optimization policy

this simple mechanism is that the value of F reflects the recent variability in the remote worker population. This ensures that the scheduling agility is suitably increased in more volatile conditions, so that relatively smaller tasks are issued, yielding greater scheduling flexibility and representing less wasted processing if a worker departs abruptly without reporting its partial result.

Determination of subsequent task-size allocations for each worker is based on the worker's current effective performance (which takes into account the size of its most recent task and the time taken to process the task, inclusive of RTT). The performance is measured relative to the performance of the other workers. The policy tolerates significant performance differentials and attempts to normalise the distribution. For example, a node that took longer to process its most recent task (relative to the mean processing time of the other workers) will be subsequently issued a smaller task. To avoid continual minor adjustments, a dead zone (± 10% margin each side of the mean) is used.

The whole system is dynamic, which implies that the *perceived* optimum deployment is also subject to change over time. To avoid sudden destabilizing changes and potential oscillation, a self-stabilizing approach is followed in which workers' allocations are adjusted *towards* the *perceived* desired value in small progressive steps. To achieve this, an adaptation factor A (discussed earlier in Section 3.1) is introduced. Empirical investigation has determined that a value of $A = 0.5$ yields good all-round performance for this specific policy. This is a compromise between higher values, which give faster adaptation when the system is stable but can cause oscillation when the system is unstable, and smaller values, which are better in unstable conditions but can be very slow to converge when the goal state follows an upward or downward trend (refer to Figure 4). The actual allocation-size adjustment (for a given worker) is thus based on AD where D is the calculated distance from the mean. For example, if the current mean processing time is 100 s and a specific worker took 50 s to process its most recent task, the distance $D = 50$, thus $AD = 25$. The task size would thus be increased by 50% to yield an expected processing time shift from 50 to 75 s for the next task. The stable converging effect of this mechanism is evident in Figure 14.

Where the performance of a particular worker is found to be below an acceptable standard and there are sufficient workers to do the job, the worker is rejected. This optimization reduces the communication overhead and the management overhead on the client. The work that would have been allocated to the slow worker can be allocated to the remaining (faster) workers. The effective value of a slow worker is greater when there are few remote workers, and diminishes as more workers become available.

https://doi.org/10.1017/S0269888906000890 Published online by Cambridge University Press

```
G_ALLOW_REMOTE_WORKER_REJECT = true
G_THRESHOLD_REJECT = 2            // The number of workers that must be present before one can be rejected
G_REMOTE_ONLY_REMOTE_SPEEDUP_THRESHOLD = 0.25
G_REMOTE_SPEEDUP_THRESHOLD = 0.5
G_LOCAL_SPEEDUP_THRESHOLD = 0.5
G_ALLOWABLE_PERCENT_DIFFERENCE_FROM_MEAN = 10
```

Figure 12 Granularity policy template values

```
if(true == m_WorkerArray[iWorkerNo].bFirstTask)     // First task sent to a worker must be of size 'g' for calibration.
{
     return G_FirstCalibration;
}
if(true == m_WorkerArray[iWorkerNo].bSecondTask)
{
     return G_InitialAllocation;                    // The initial allocation rules are used to determine the second task sent to a worker.
}
if(iNumberOfRemoteWorkers >= iThresholdReject && true == b_AllowRemoteWorkerReject)
{                                                   // Determine whether to reject a worker
     if(dSpeedupRelativeToLocal == -1.0)            // -1.0 signals that the local worker is not currently executing
     {
          if(dSpeedupRelativeToRemote < dRemoteOnlyRemoteSpeedupThreshold)
          {
               return G_RejectWorker;               // Worker is VERY slow relative to other remote workers
          }
     }
     else                                           // Local worker IS running
     {
          if(dSpeedupRelativeToRemote < dRemoteSpeedupThreshold &&
                    dSpeedupRelativeToLocal < dLocalSpeedupThreshold)

          {
               return G_RejectWorker;               // Worker is slow relative to other remote workers AND local worker
          }
     }
}
// Determine task-size adjustment
if(m_WorkerArray[iWorkerNo].lElapsedTime_MostRecent_MilliSeconds >
     (lMeanProcessingTime + (lMeanProcessingTime * iAllowablePercentDifferenceFromMean / 100)))
{
     return G_MakeFinerGrained;                     // Worker (most-recent task) took longer than the mean
}

if(m_WorkerArray[iWorkerNo].lElapsedTime_MostRecent_MilliSeconds<
     (lMeanProcessingTime - (lMeanProcessingTime * iAllowablePercentDifferenceFromMean / 100)))
{
     return G_MakeCoarserGrained;                   // Worker (most-recent task) took less time than the mean
}
return G_LeaveGranularityAsIs;                      // Worker response-time currently in the Dead-Zone (close to mean)
```

Figure 13 Outline of the granularity policy logic (some detail omitted)

For each worker, the policy responds with a decision to increase, decrease or leave unchanged the task size; or to reject the worker. The actual amount of adjustment is calculated externally to the policy. This simplifies the policy implementation and helps maintain the overall design goal of inserting numerous *simple* policies. The default template preferences for the policy are shown in Figure 12. The policy logic is presented in Figure 13.

Figure 14 provides an example of the granularity policy behaviour in a scenario where workers arrive gradually, joining at an approximately 15 s inter-arrival interval. Workers had low stable background workloads throughout the experiment. The flexibility factor (F) was initialised to 15 and was automatically incremented each time a worker arrived to reflect the extent of variation in the worker population. This has the effect of increasing the scheduling flexibility by enabling smaller fractions of the remaining work to be issued as initial tasks to newly-arriving workers, thus allowing finer-grained optimization. After initial allocation the policy adapts the granularity of subsequent tasks towards the mean over all workers to ensure that the workload is evenly distributed.

Stable adaptation is evidenced by the way the task allocations converge quickly towards the mean size (which is itself dynamic) but do not tend to oscillate around it. Two mechanisms directly contribute to the stability achieved.

1. The adaptation factor A (set at 0.5) governs the way the allocated task sizes home-in progressively towards the mean task size.

https://doi.org/10.1017/S0269888906000890 Published online by Cambridge University Press

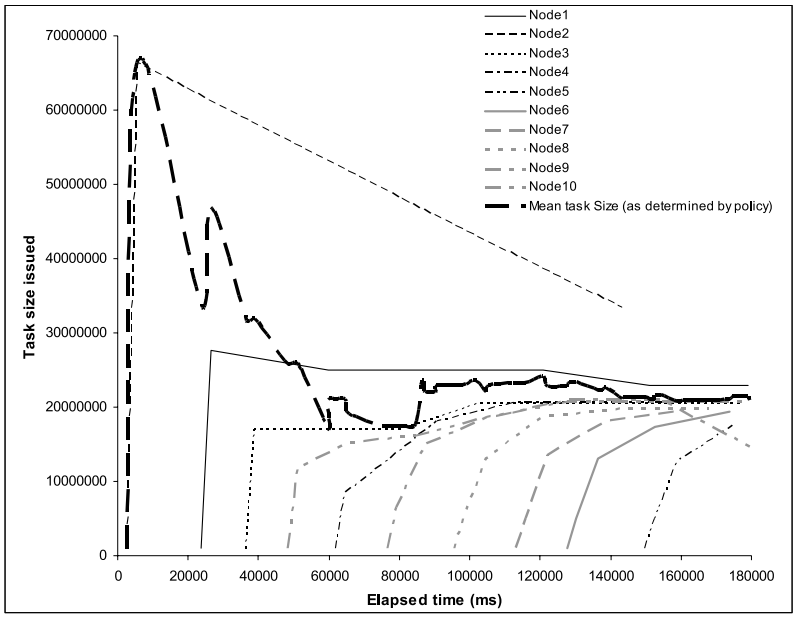


Figure 14 An example of the behaviour of the granularity policy when the worker population increases dynamically

2. The dead zone halts the adaptation process (for a specific worker) as its allocated task size becomes sufficiently close to the mean.

The policy managed to adapt task allocations sufficiently that it could utilize each worker effectively, achieving an overall speedup of 0.593 against a base-line configuration comprising 10 constantly-available workers. The gradual-arrival scenario is particularly challenging in terms of scheduling optimization; the accuracy of the adaptation and the extent of the performance results achieved powerfully illustrate the feasibility and benefits of the flexible deployment approach for dynamic environments.

6 Conclusion and further work

This paper has presented an empirical investigation into ways in which autonomic behaviours can be embedded into parallel applications that target loosely-coupled environments such as clusters and Grids. These environments have potentially highly dynamic and heterogeneous natures, so a strategy of continually re-evaluating the deployment of tasks across worker nodes is proposed with the goal of maintaining high performance despite the environmental volatility.

A prototype application has been developed, in which self-managing behaviour is achieved through the use of several policies, each responding to specific aspects of environmental or contextual state in the governance of a clearly-defined dimension of behaviour. A high degree of flexibility is required to ensure that an executing application is sufficiently responsive to a wide range of ambient conditions. In this regard, several mechanisms have been implemented to facilitate dynamic monitoring of environment and context, including: node discovery; detection of failed workers; calibration of network connections; and calibration of effective worker-node performance.

The policies have been equipped with various self-stabilization features including the incorporation of behavioural bounds, dead zones and gradual adaptation to help ensure that they continuously track their goal states in a convergent manner despite considerable environmental perturbations.

The prototype application has been used to explore the feasibility of the strategy and the extent of the associated benefits. The design of three policies has been discussed. The behaviour of these policies has been described in terms of the ways they react to contextual and environmental stimuli.

https://doi.org/10.1017/S0269888906000890 Published online by Cambridge University Press

Empirical results demonstrating the adaptive behaviour of the policies have been presented. Sample RTT-measurement-optimization policy results demonstrate bounded self-optimization, and the effectiveness of using a dead zone to ignore noise-level changes in communication latency. Sample granularity-policy results demonstrate:

- the achievement of deployment flexibility in the way in which the policy holds back sufficient work to be able to utilize new workers (and the way in which the flexibility factor is itself adapted to reflect variability in the workforce);
- discovery and self-configuration through the automatic incorporation of newly-arriving workers;
- stable, convergent self-optimization through the subsequent gradual adjustment of the deployment function such that the response times of the worker population are normalized.

By combining the behaviour of several policies (there are actually five implemented), quite sophisticated self-managing behaviour is achieved. The application is able to adjust several dimensions of its behaviour sufficiently quickly to be effective, yet in a progressive manner, avoiding destabilizing sudden shifts.

References

Allen, C. B. 2005 Parallel simulation of lifting rotor flows: a wake capturing study. In *Proceedings of the DCABES and ICPACE Joint Conference on Distributed Algorithms for Science and Engineering*. University of Greenwich: CMS Press, pp. 23–30.

Ananthanarayanan, R., Mohania M. & Gupta, A. 2005 Management of conflicting obligations in self-protecting policy-based systems. In *Proceedings of the 2nd International Conference on Autonomic Computing (ICAC)*. Seattle, WA: IEEE, pp. 274–285.

Badr, N., Taleb-Bendiab, A., Randles, M. & Reilly, D. 2004 A deliberative model for self-adaptation middleware using architectural dependency. In *Proceedings of the 15th International Conference on Database and Expert Systems Applications (DEXA 2004)*. IEEE, pp. 752–756.

Baiardi, F., Chiti, S., Mori, P. & Ricci, L. 2001 Integrating load balancing and locality in the parallelization of irregular problems. *Future Generation Computer Systems 17*(8), 969–975.

Barrett, R., Maglio, P., Kandogan, E. & Bailey, J. 2004 Usable autonomic computing systems: the administrator's perspective. In *Proceedings of the 1st International Conference on Autonomic Computing (ICAC 2004)*. New York: IEEE Computer Society, pp. 18–25.

Clementi, E. & Corongiu, G. 1999 Early parallelism with a loosely coupled array of processors: the ICAP experiment. *Parallel Computing 25*(13–14), 1583–1600.

Du, Z. H., Wang, H., Yang, F. & and Li, S. L. 2005 Pattern-based parallel model to decide suitable-grained parallelism for cluster computing. In *Proceedings of the DCABES and ICPACE Joint Conference on Distributed Algorithms for Science and Engineering, August 2005*. University of Greenwich: CMS Press, pp. 191–194.

Goscinksi, A. 1999 Finding, expressing and managing parallelism in programs executed on clusters of workstations, *Computer Communications 22*, 998–1016.

Gottlieb, S. 2001 Comparing clusters and supercomputers for lattice QCD. *Nuclear Physics B (Proc. Suppl.) 94*, 833–840.

Grandison, A. J., Gallea, E. R., Patel, M. K. & Ewer, J. A. C. 2005 Parallel CFD based fire modelling on conventional office based PCs. In *Proceedings of the DCABES and ICPACE Joint Conference on Distributed Algorithms for Science and Engineering, August 2005*. University of Greenwich: CMS Press, pp. 43–46.

Hendrickson, B. 2000 Load balancing fictions, falsehoods and fallacies. *Applied Mathematical Modelling 25*, 99–108.

Hobbs, M. & Goscinski, A. 2000 A The GENESIS parallelism management system employing concurrent process-creation services. *Microprocessors and Microsystems 24*, 415–427.

Hu, Q. & Zou, J. 2005 Nonoverlapping domain decomposition methods for three-dimensional maxwell system in non-homogeneous media. In *Proceedings of the DCABES and ICPACE Joint Conference on Distributed Algorithms for Science and Engineering*. University of Greenwich: CMS Press, pp. 17–21.

IBM. 2004 An architectural blueprint for autonomic computing. IBM Autonomic Computing White Paper. Available at: `http://www-03.ibm.com/autonomic/pdfs/ACBP2_2004-10-04.pdf`.

Ibrahim, M., Telford, R., Dini, P., Lorenz, P., Vidovic N. & Anthony, R. 2004 Self-adaptability and man-in-the-loop: a dilemma in autonomic computing systems. In *Proceedings of the 2nd International Workshop on Self-Adaptable and Autonomic Computing Systems — SAACS '04 (DEXA 2004), Zaragoza, Spain*, September. IEEE, pp. 722–729.

https://doi.org/10.1017/S0269888906000890 Published online by Cambridge University Press

Jin, X. & Liu, J. 2004 Characterizing autonomic task distribution and handling in Grids. *Engineering Applications of Artificial Intelligence 17*(7), 809–823.

Kephart, J. O. & Chess, D. M. 2003 The vision of autonomic computing. *Computer 36*(1), 41–50.

Koehler, J., Giblin, C., Gantenbein, D. & Hauser, R. 2003 On autonomic computing architectures. Research Report (Computer Science) RZ 3487(#99302), IBM Research (Zurich).

Maglio, P., Campbell, C. & Kandogan, E. 2005 On the need for negotiation in policy-based interaction with autonomic computing systems. In *Proceedings of the 2nd International Conference on Autonomic Computing (ICAC)*. Seattle, WA: IEEE, pp. 356–357.

Shum, K. H. 1999 Effective fault tolerance for agent-based cluster computing. *Journal of Systems and Software 48*(3), 189–196.

Weng, J., Miao, C. & Goh, A. 2004 Dynamic negotiations for Grid services. In *Proceedings of the 1st International Conference on Autonomic Computing (ICAC 2004)*. New York: IEEE Computer Society, pp. 296–297.

https://doi.org/10.1017/S0269888906000890 Published online by Cambridge University Press

https://doi.org/10.1017/S0269888906000890 Published online by Cambridge University Press