Engineering Emergence for Cluster Configuration

Richard John ANTHONY
Department of Computer Science, University of Greenwich
Greenwich, London, SE10 9LS, United Kingdom.

ABSTRACT

Distributed applications are being deployed on ever-increasing scale and with ever-increasing functionality. Due to the accompanying increase in behavioural complexity, self-management abilities, such as self-healing, have become core requirements. A key challenge is the smooth embedding of such functionality into our systems.
Natural distributed systems such as ant colonies have evolved highly efficient behaviour. These emergent systems achieve high scalability through the use of low complexity communication strategies and are highly robust through large-scale replication of simple, anonymous entities. Ways to engineer this fundamentally non-deterministic behaviour for use in distributed applications are being explored.
An emergent, dynamic, cluster management scheme, which forms part of a hierarchical resource management architecture, is presented. Natural biological systems, which embed self-healing behaviour at several levels, have influenced the architecture. The resulting system is a simple, lightweight and highly robust platform on which cluster-based autonomic applications can be deployed.

Keywords
Dynamic Cluster Management, Self-Healing, Emergence, Scalability, Fault-Tolerance, Layered Architecture

1. INTRODUCTION

A cluster management scheme to support autonomic applications is described. The scheme is designed specifically to support the significant subset of such applications that can be said to be LAN-scoped, that is, are deployed within the geographical area of a LAN and can thus enjoy the benefits of such technology – low latency and broadcast communication being specific examples.

LAN-scoped autonomic applications are deployed in a wide-range of domains; examples include aircraft control systems, industrial plant management and monitoring systems, and high-performance computing applications distributed over processor pools. These application domains have some common requirements which include: robustness, the ability to reconfigure dynamically; scalable deployment platforms; stability despite configuration change; efficiency in the use of systems resources, especially in relation to the number of messages transmitted as scale increases; and low communication latency because the applications have a real-time aspect.

Clusters of loosely-coupled processors provide a suitable platform for this class of distributed application. Specifically clusters are robust (because of their natural ability to provide redundancy), scalable (because of their ability to be incrementally expanded), efficient (because resource-pull scheduling, in which idle processors request tasks, and/or load sharing can be implemented), offer suitable communication modes (broadcast and multicast mechanisms are a useful technique to cut down the number of messages and the communication latency when many components interact). Clusters are also highly cost-effective because the physical resources they employ are often already deployed as part of a general purpose system. The ‘cluster’ can be a logical subset of the physical computers in a LAN, selected by some criteria (typically that they have low load).

Implementation examples of cluster-based autonomic systems include a multi-agent system for shipboard automation [1], in which a number of systems are interconnected and whilst retaining autonomy, cooperate to deliver fault-tolerant, adaptive behaviour; and [2], a ubiquitous museum information system which provides real-time modification of museum information to suit visitors’ specific preferences.

The automatic deployment of diverse applications in non-dedicated systems requires the dynamic creation and maintenance of appropriately-sized clusters to ensure effective and appropriate use of resources.

This paper proposes that the autonomic behaviour of systems should be *layered throughout the entire system*, including the software platforms and core services that support higher-level autonomic applications. [3] Suggests that autonomic computing systems can only be controllable if they consist of components with limited capabilities and finite internal state-spaces. Extending the autonomic behaviour to the core services, such as cluster management, creates additional opportunities for the provision of self-organising and self-healing, removing the burden for entire provision at the application layer.

A hierarchical implementation approach enables functionality to be spread across several components arranged in layers. Thus individual components can be less complex, as they ‘inherit’ or use the services of components in lower layers. This in turn makes components simpler to develop and likely to be more reliable, and therefore improves the overall scalability, robustness and stability of systems.

Natural emergent systems employ self-healing at many levels, and in many contexts, simultaneously. Mammalian immune systems operate at the cellular level and self-heal by ‘learning’ to distinguish dangerous invaders, evolving ways to defend against them [4, 5]. Mammals also employ self healing at the level of individual organs, such as the way the human liver can recover from alcohol poisoning. Mammals self-heal at the animal level, for example by resting an injured limb, or by eating medicinal herbs. The combination of self-healing at so many different levels enables the system to cope with a very-wide range of disturbances, deploying the most appropriate mechanism(s) for a particular situation.

As with transparency, which should pervade all layers of software systems, i.e. it should be 'designed in' rather than 'built on' [6], this paper proposes that self-healing should be an intrinsic property throughout the system, as in the evolved biological systems.

Many highly successful distributed systems are found in nature. Systems such as ant colonies have evolved highly efficient behaviour, including the ways in which the actors communicate. These emergent systems deploy low complexity communication strategies, enabling them to achieve high scalability. A high level of robustness is achieved through the large-scale replication of simple, anonymous entities which act autonomously.

Distributed computer applications typically have the same non-functional requirements as natural biological systems, such as the need for efficiency and low-complexity communication. In particular however, scalability and robustness are increasingly important as more and more systems with global and ubiquitous scope are launched. Designing to simultaneously meet demands of high scalability and high robustness can lead to conflicts.

Communication design is one of the most critical aspects of distributed application design. With traditional designs, the more robust an application is, the greater the communication intensity tends to be (because of, at least some of, checkpointing activities, status messages, updates of replicas and acknowledgement messages). This in turn impacts on scalability. The amount of stored-state is also often greater in robustly designed applications. Excessive communications can degrade performance. Network bandwidth is also a precious commodity that should be conserved where possible. The latency of decisions is often extended as the number of communication partners and the amount of state exchanged is increased. For these reasons, it is important that highly efficient communication strategies are employed, in accordance with the specific requirements of a particular distributed application.
Due to issues arising from their design, including the communications strategies, many implemented distributed applications fail to fully meet all of their non-functional requirements; compromises occur.

The design of the cluster management scheme presented in this paper is inspired by the self-healing characteristics of natural systems described above (i.e. it has a hierarchical architecture that deploys self-managing behaviours at several layers which cooperate to provide a highly adaptive and robust framework). The system layer and cluster layer have self-stabilising emergent behaviours which continually adjust the system towards the desired configuration despite disturbances. The autonomic applications which are deployed at the application layer will have their own self-healing behaviour.

The rest of the paper is organised as follows: section 2 discusses the use of clusters in resource management; section 3 discusses emergence; section 4 provides an outline of the resource management architecture; section 5 identifies the emergence aspects of the design; section 6 provides an overview of the operation of the simulation model; section 7 evaluates the performance of the system and cluster layers; and section 8 presents the conclusions.

2. CLUSTER-BASED RESOURCE MANAGEMENT

Within most large organisations there exist research groups and individuals who require greater processing capacity than is currently available to them. Researchers performing computationally-intense simulations involving for example CFD, Fourier transforms, and the like must either acquire sufficient dedicated resource (at high cost) or suffer the relatively slow response-times afforded by the 'standard' provision (each user's desktop).

Meanwhile, these organisations tend to have a massive unused computing capacity. This occurs because at any given moment many of its computers lie idle whilst a large fraction of those that are in use are underutilised [7, 8, 9]. This resource is widely distributed in staff offices and computer laboratories. One way to reclaim some of this unused resource is to encompass these computers within a cluster-based resource management system.
Clusters of workstations offer a flexible platform on which a wide variety of distributed applications and services can be based. The main reasons for the popularity of the cluster approach are its extensibility, efficient use of resources and better cost/performance effectiveness (when compared to large-scale multi-processor machines [10, 11, 12, 13]), robustness (see for example the cluster-level recovery scheme described in [14]) and load sharing [15, 16].

Several cluster architectures have been specifically designed to support parallel applications [10, 17] or sub-tasks such as parallel query processing, as in [11]. [16] Describes a scheme in which load distribution is achieved by distributing the middleware components over the cluster. In [18], a scheme in which the server-side of applications are run over a cluster but the client side executes at the originator workstation, is described.

Non-dedicated clusters for processing coarse-grained parallel tasks can be built using existing general-purpose computers which have individual owners [12]. Such non-dedicated clusters reuse existing computers and only execute cluster tasks when their owner-initiated workload is low.

Non-dedicated clusters must be capable of dynamic self-management because the actions of the individual-computer owners are unpredictable. Users who have specifically allocated computers tend to leave them powered on throughout the working day (and even overnight), logging in/out as required. Users of shared laboratory-based computers might reboot a computer at the start or end of a use session.

A cluster whose coordinator node (or any other) is rebooted by its local user (or otherwise 'fails') must be able to recover in a way that is transparent to the cluster-level task.

3. AN OVERVIEW OF EMERGENCE

The science of emergence is described in [5, 19, 20]. Emergence is the term used to describe a higher-level state or pattern or other behaviour that arises from the interaction of lower-level components. The higher-level behaviour cannot be predicted by examining the individual components or their behaviour in isolation.

The term ‘engineered emergence’ is used in this paper to describe the purposeful design of interaction protocols so that a predictable, desired outcome is achieved at a higher level (i.e. emerges), although at lower levels the specific state of individual components at any moment cannot be predicted. For (very simple) example, consider a pair of processing nodes each capable of providing a given service. In addition each node sends periodic messages to inform the other node of the sender’s state {*active service provider*, *standby*}. It is straightforward to imagine how such a scheme can be tuned (typically employing random numbers in some way to break symmetry) to automatically ensure that a single node provides service at all times that at least one of them is capable to do so. The status of individual nodes at any moment cannot and need not be predicted, but the overall status of the system is predictable within the system’s stated operational envelope (at least one node is ‘healthy’).

Thus emergence offers significant potential to the developers of distributed systems. It can be witnessed in nature that many successful systems that exhibit complex behaviour are made up of large numbers of very simple entities, each exhibiting very simple behaviour and having no global system knowledge [21]. These self-organising systems have evolved solutions for problems similar to those that we face when designing our systems.

Of particular interest is the pheromone exchange communication employed in (for example) Harvester Ant colonies, in which different pheromones have different meanings and anonymous ants pay attention to the frequency and strength of the messages they receive [5]. Ants autonomously determine their behaviour over the short-term based on these pheromone exchanges. Individuals have low intelligence and only have a local perspective. The overall colony behaviour is highly ‘intelligent’ achieving sophisticated goals such as nest building and defence, and cooperative foraging for food

Emergent systems are self-healing in many contexts, including in the most literal sense: Immune systems, operating at the cellular level, ‘learn’ to distinguish dangerous invaders, and evolve ways to defend against them [5].

4. THE CLUSTER MANAGEMENT SCHEME

Overview

The functionality required to provide reliable, scalable and efficient cluster-based task execution has been mapped onto a three-layer resource management architecture, as shown in figure 1.

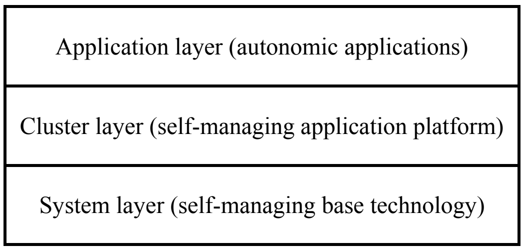

Application layer (autonomic applications)
Cluster layer (self-managing application platform)
System layer (self-managing base technology)

Figure 1. The resource-management architecture.

The resource consists of a pool of available workstations. At the system-layer, nodes cooperate to maintain a single system-wide coordinator. This coordinator is responsible for dynamically creating clusters of specified size, on demand. The design of the system-layer ensures that clusters are created quickly and efficiently, even in large systems. This layer is based on a highly efficient election algorithm which combines deterministic and non-deterministic behaviours to achieve self-healing and adaptation whilst remaining very stable, and because it uses a very small number of messages, is very scalable.

A cluster is a subset of the workstations in the system, assembled to process a specific task. Nodes that are available to join clusters, i.e. have low load and are not already a member of a cluster, join the system-level pool. Nodes are then recruited from this pool into clusters, as required. Tasks that require execution on a cluster make a request to the system-level coordinator for a cluster to be created. The cluster-level adapts the election algorithm on which the system-layer is based, to maintain a single coordinator per cluster. The cluster coordinator manages the execution of requests sent to the cluster.

The application layer is concerned with requesting cluster creation, submitting tasks to the cluster for execution, overseeing the execution and ensuring that it completes successfully.

Broadcast communication is used within both the system-layer and cluster-layer so that there is no need for individual non-coordinator nodes to keep track of the identity of their cluster coordinator, or for the coordinator to keep track of the identities of individual cluster members. This stateless approach facilitates simple role allocation, provided by the election algorithm, without the complexity of additional informational updates each time a change in membership or role occurs.

Discussion

This paper is primarily concerned with the design of the system layer and the cluster layer. The functional requirements of these layers are: maintain exactly one system-level coordinator to manage cluster creation; create clusters on demand, of specified size; and maintain exactly one coordinator per cluster to manage the cluster (i.e. receive tasks, distribute over the cluster, collect results, transmit to task originator and disband the cluster). The non-functional requirements are: stability; high scalability; high robustness; efficiency, especially in terms of communications intensity; and low-latency.

Externally deterministic behaviour is needed to ensure the functional requirements are fully met. However, it can be witnessed that natural distributed systems (which are inherently non-deterministic) have evolved better strategies for achieving non-functional requirements very similar to those we are presented with when developing distributed applications. The design approach taken in this work is based on the view that a lot of the (internal) determinism built into our systems (at great expense) is redundant. What is important is that the systems behave deterministically overall. It is not always important that every single step is deterministic. The design and development of the cluster management system thus contributes to the exploration of the extent to which the benefits of non-deterministic design can be harnessed within successful distributed computer applications.

An election algorithm based on emergent behaviour forms the system layer of the architecture. Nodes adopt one of four states as shown in the state-transition diagram (figure 2). The algorithm achieves very high scalability because most of the nodes remain in the idle state, in which they are completely passive with respect to communication. Slave nodes have the role of monitoring the presence of the master, and upon its failure, of electing a replacement. A small pool of slaves is sufficient to achieve this behaviour reliably so elections are very efficient. Idle and slave nodes count the number of periodic transmissions from slave nodes over a time interval and compare them to a pair of threshold values. An idle node whose local count of slave messages is below the lower threshold elevates to slave state. A slave node whose local count of slave messages is above the upper threshold demotes to idle status.

A random component in the local time interval, over which slave messages are counted, and randomness in end-to-end messaging latency breaks the symmetry which, in conjunction with a dead-zone between the lower and upper threshold values, ensures that the non-deterministic idle-slave interaction is stable, although there can be some fluctuation in the size of the slave-pool.

The election of a master (the coordinator of either the system layer or of a specific cluster) must be deterministic, as there must be exactly one node elected. To ensure this, the candidate state is introduced as an intermediary between the slave and master states, to facilitate a form of implicit negotiation. A slave node that times-out three times consecutively whilst waiting for periodic master messages elevates itself to the candidate state. It then sends a candidate message containing its node ID. This message is interpreted by the remaining slaves as "I have noticed the lack of a master and will elevate to that role if I do not hear from any nodes with a higher ID". The node then waits a short time period for responses. On receipt of the candidate message, slave nodes that have a higher ID now elevate to candidate status and they too send a candidate message. Candidate nodes that receive candidate messages from higher ID nodes demote to slave status. After a short time only the highest ID candidate should remain and, on timeout, it elevates to master. The node immediately transmits a master message which causes any remaining candidate nodes to demote immediately to slave state and all slave nodes to reset their timers and revert to monitoring the presence of the master.

The symbol n represents the total number of nodes in the system.

The symbols m, c, s, i, represent the number of nodes with master status, candidate status, slave status and idle status respectively, other than during elections.

The symbols m_E, c_E, s_E, i_E, are used to represent the number of nodes with master status, candidate status, slave status and idle status respectively, during elections.

At all times except during elections:

$$n = m + c + s + i \tag{1}$$

During elections:

$$n = m_E + c_E + s_E + i_E \tag{2}$$

The communication intensity of the algorithm is very low. An election begins when a slave times-out and elevates to candidate status because it has not received any master messages. Thus, immediately before an election begins:

$$i \geq 0,\ s \geq 1,\ c = 0 \text{ and } m = 0 \tag{3}$$

Eq. (3) states that at least one slave node must exist in order for an election to begin. This could be the only node in the system.

During an election:

$$i_E \geq 0,\ s_E \geq 0,\ c_E \geq 1 \text{ and } m_E = 0 \tag{4}$$

where:

$$1 \leq c_E \leq s \tag{5}$$

Eq. (4) states that at least one candidate node must exist during an election. This could be the only node in the system.

With respect to i_E idle nodes and s_E slave nodes (those that remain slaves throughout the election), the communication complexity is zero (as they do not participate in the election).

Eq. (5) states that at least some, but not necessarily all of the nodes that are slaves prior to the election elevate to candidate state during an election. The value of c_E depends on the ID of the slave that first elevates to candidate status, relative to the IDs of the remaining slaves. In the worst case, the slave with the lowest ID elevates first and the remaining slave nodes respond to the candidate messages in ascending order. Conversely, if the slave with the highest ID elevates first, it will suppress the elevation of other slaves. On average: $c_E = s/2$, implying that $s/2$ candidate messages are typically generated during an election. The election is completed when the new master node sends the first of its periodic master messages.

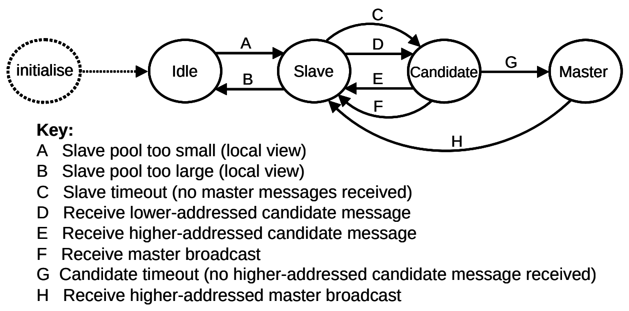

initialise
Idle
Slave
Candidate
Master
A
B
C
D
E
F
G
H
Key:
A Slave pool too small (local view)
B Slave pool too large (local view)
C Slave timeout (no master messages received)
D Receive lower-addressed candidate message
E Receive higher-addressed candidate message
F Receive master broadcast
G Candidate timeout (no higher-addressed candidate message received)
H Receive higher-addressed master broadcast

Figure 2 State transitions at the system level

Therefore election complexity is:

$$O(s/2 + 1) \tag{6}$$

which is independent of the system size. The typical slave-pool size is between two and four nodes:

$$2 \le s_{TYPICAL} \le 4 \tag{7}$$

A typical election therefore (from Eq. (6) and Eq. (7)) requires only two or three messages (for any system size).

The normal mode communication (i.e. in the absence of elections) is also highly efficient. In this mode:

$$i \ge 0, s \ge 0, c = 0 \text{ and } m = 1 \tag{8}$$

Eq. (8) states that exactly one master node must exist during normal operation. This could be the only node in the system.

With respect to *i* idle nodes the communication complexity is zero (as no messages are sent). Slaves and *the* master node transmit messages at slow periodic rates (intervals of 10 and 5 seconds respectively). The normal mode communication complexity is thus:

$$O(s/10 + 1/5) \tag{9}$$

messages per second, independent of the system size. The typical communication cost of running the algorithm, outside elections, is thus approximately 0.5 messages per second (from Eq. (7) and Eq. (9)). For the system-level communication all messages contain only a single data byte to indicate message type, and so they are of the minimum frame size. For a Fast Ethernet network this equates to less than 3 millionths of the bandwidth. The emergence-inspired design is very efficient.

The role of the system-level coordinator is to create clusters of specified size on demand. The interaction that occurs to achieve this is illustrated in figure 3.

Several message types are used. CreateCluster(*n*) is a directed broadcast from an external node requesting that a cluster of size *n* be created. The use of directed broadcast avoids the need to know the address of the system-level coordinator, which is dynamically elected. InviteMembershipBids(*k*) is a broadcast message inviting nodes to join cluster *k*. The value of *k*, the cluster ID, is chosen by the system-level coordinator.

To reduce the total number of messages needed to create a cluster, a 'delayed-bids' mechanism has been devised. Nodes that are not currently members of clusters reply to an InviteMembershipBids(*k*) message by sending a unicast MembershipBid(*k*), after waiting a short, random *delay* time.

The system-level coordinator accepts the required number of bids by sending unicast AcceptBid(*k*, *role*) messages. The *role* parameter indicates the node's initial role within the newly formed cluster. The first node accepted initially coordinates the new cluster (saving the cost of an initial election), others are given the *idle* role. Once the appropriate number of acceptances have been issued, the system-level coordinator broadcasts a StopBids(*k*) message, which has the effect of cancelling any outstanding unsent MembershipBid(*k*) messages. The performance benefit of the delayed bids mechanism is evaluated in section 7.

The delayed bids mechanism is more efficient than bidding protocols such as the Contract Net [22] because the delayed response reduces the total number of messages sent, and also the burst of communication (when all nodes respond within a very short time-frame) which causes congestion and thus delay. The random delay values used by bidding nodes effectively replaces the selection function. The selection role (of bidding / allocation protocols such as Contract Net) is reduced to simply counting the number of replies needed and responding to them, thus reducing computational complexity at the coordinator.

A unicast CreateClusterAck(*k*) message is used to signal the ID of the cluster created to the request originator. The cluster ID relates directly to the *port* number the cluster uses for its private communication. The cluster originator can thus locate and communicate with the cluster coordinator using a directed broadcast and the cluster port number. This approach avoids the need for the application-level cluster-deployed application to be aware of the specific address of the cluster coordinator (which is subject to change, for example due to failure of the original coordinator).

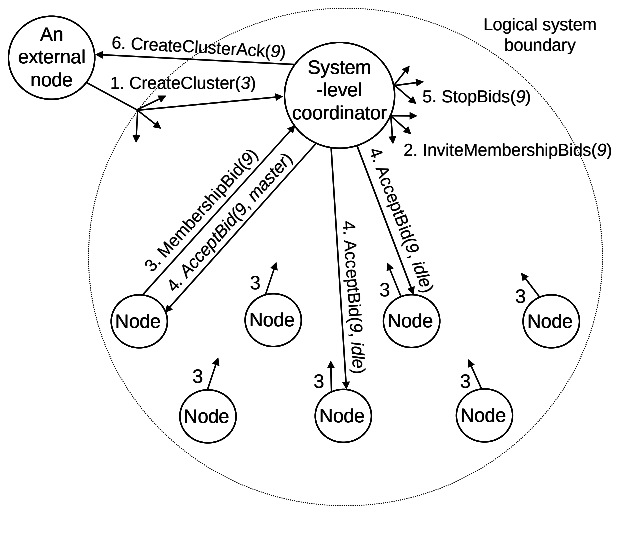
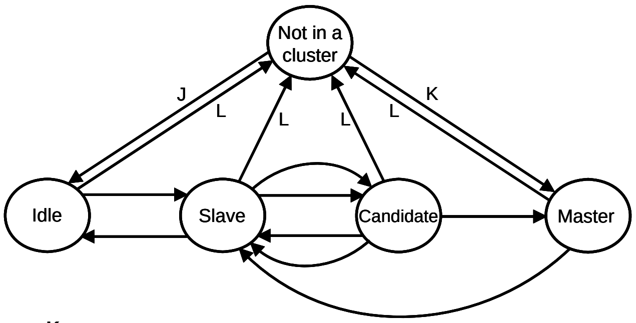


Figure 3. The interaction that occurs to create a cluster of 3 nodes.

For each cluster created, a coordinator must be maintained to manage the execution of application-level requests sent to the cluster. Thus the cluster-level reuses the election algorithm described above, extending the highly efficient and robust design up to this second level of the architecture. The state-transition behaviour at the cluster-level is modified to allow for the fact that nodes do not always have cluster membership. Nodes retain a role at the system-level regardless of whether they are a member of a cluster or not. Figure 4 illustrates the cluster-level modifications.

The first node assigned to each cluster is designated by the system coordinator to be the coordinator of the cluster. From that point on the cluster operates independently and maintains its coordinator.

The system level coordinator is not concerned with subsequent changes that occur within the cluster. This cluster-level independence contributes to the simplicity, and thus to the robustness and efficiency of the architecture.



Key:
J. Receive AcceptBid message from system coordinator, indicated initial cluster-level role is *Idle*
K. Receive AcceptBid message from system coordinator, indicated initial cluster-level role is *Master*
L. Receive DisbandCluster message from cluster coordinator

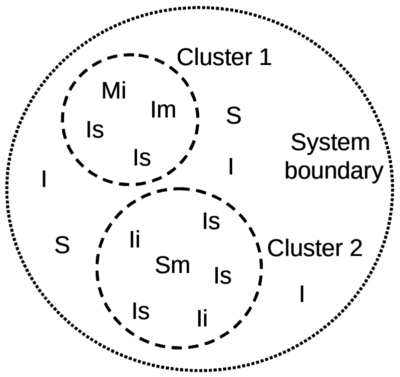
Figure 4. Modified state transition diagram for the cluster level.

All aspects of actual task execution on the cluster are the concern of the application layer. This separation allows the cluster layer operation to be highly efficient.

The simple architecture incorporates self-adaptability at each layer, resulting in a highly robust, scalable and flexible resource configuration system. The cluster-layer re-uses and adapts the highly efficient design of the system-layer to the management of clusters. As with the system layer, the cluster layer utilises non-deterministic behaviour to achieve highly scalable and robust behaviour, which is stable and externally appears deterministic because it self-adapts and self-heals such that a single coordinator is maintained for each cluster. This coordinator is responsible for cluster-level activities such as the replacement of failed nodes and the release of members when the cluster's host application terminates. In addition to managing its cluster, the cluster coordinator may take on a role at the application-level, depending on the nature of the application deployed on the cluster. It may for example, provide coordination of specific activities such as managing transactions, identify the master instance of a replicated service, or it may handle communication with external components.

The application layer comprises autonomic applications which require a stable and reliable cluster platform on which to operate. Clusters are created dynamically to suit the needs of such applications.

A typical configuration snapshot is illustrated in figure 5. Note 1. there is a coordinator at the system layer and also for each cluster, 2. nodes within clusters retain system-level roles.



Key: Upper-case indicates system-level role, lower-case indicates cluster-level role. M = master (coordinator), S = slave, I = idle (candidate state only occurs during elections)

Figure 5. A typical system configuration containing two dynamically created clusters, each with their own dynamically-elected coordinator.

5. EMERGENCE ASPECTS OF THE DESIGN

The design has been inspired by natural systems, especially in terms of communication and interaction protocols. In natural emergent systems these tend to be simple, consisting of a small number of simple rules.

Communication at both the system level and the cluster level is loosely modelled on pheromone communication systems found in natural distributed systems such as ant colonies [5, 20]. This yields low interaction and communication intensity, although much of the internal behaviour is non-deterministic. It cannot be predicted which role a particular node will take on at any given time, at either level.

However, a stable pattern (configuration) is certain to emerge. I.e. the overall behaviour (described at a higher level) is deterministic despite the non-deterministic behaviour at the level of individual nodes at any specific moment.

The pheromone-based communication strategies involve low numbers of simple messages which contributes significantly to high scalability. The messages have individually low value so no recovery action is needed if they are lost, enhancing robustness. The election algorithm that has been employed has been purposefully designed to tolerate high levels of (non-recovered) message loss (see performance evaluation in section 7, and also [23]).

As is common in the natural systems, the main non-deterministic interaction (the idle-slave interaction) is regulated by negative feedback and relies on several sources of randomness to break the symmetry (and thus avoid oscillation that can occur when many nodes take the same action at the same time). This, combined with the use of negative feedback, ensures stability.

In keeping with the minimal state storage used in natural systems, very little state is required for nodes to operate at both levels. In addition to the node's IP address, which is used as its system-wide unique NodeID, a node's state consists simply of five integers:

- *SystemLevelRole* {Master, Candidate, Slave, Idle};
- *SystemLevelCountOfSlaveMessages* (used in the idle-slave interaction);
- *ClusterID* (-1 indicates 'not-in-a-cluster');
- *ClusterLevelRole* {Master, Candidate, Slave, Idle};
- *ClusterLevelCountOfSlaveMessages* (used in the idle-slave interaction);

Some additional state will be held at the application-level, concerning the execution of the task on the cluster.

As with pheromone-exchange, almost all of the information transmitted in messages is used to make decisions upon message receipt. Very little transmitted information is retained as state information.

Node's state mainly arises as the result of autonomous decisions based on the node's local system view. As with pheromone-exchange systems, the frequency of communication events is itself a source of information. For example the idle-slave interaction (at both levels) requires that nodes maintain a count of slave messages received over a short time interval. Such messages have no content other than the message-type identifier (one byte).

A further simple innovation borrowed from natural systems is the concept of randomly delaying interactions. In systems such as an ant colony, individual actors interact at random intervals. An external stimulus, such as an attack by another colony, is not communicated immediately to all actors – it ripples through. Once sufficient actors have responded, the propagation is ceased. Although this delay increases latency, it can also be used a means of dramatically reducing the number of messages actually required to perform some function. This approach has been used in the cluster management system when forming clusters. Details of the delayed bid implementation, and its effectiveness in reducing message numbers, are provided in section 7.

It was desired to retain the autonomy and anonymity of nodes to as great an extent as possible, as these contribute significantly to self-organisation and robustness, whilst keeping the number of messages low. Clusters of a specified size must be created on demand, without the system-level coordinator having to specifically identify each node, and without costly inter-node negotiation or many message rounds. Nodes do not cache the addresses of others at any time. Unicast reply addresses are retrieved from received messages so there is no need to remember, for example, the identity of the coordinator (which is subject to change). Cluster members identify one-another simply by the cluster ID, which can be translated directly into a cluster-private port number.

6. OVERVIEW OF MODEL OPERATION

The model is based around an array of node details. For each node in the array, a linked-list of delivery-pending messages is maintained.

Both unicast and broadcast messages are used. At the point of generation, a probability distribution is used to determine if the message should be dropped, to simulate message loss or corruption. Messages deemed to have been lost or corrupted are not actually generated.

Messages, once generated, are placed in the message-list for each recipient node. Data held concerning each message includes the timestep at which it should be delivered to the recipient. The model uses a 1ms timestep. The delivery timestep is offset into the future by a random transit delay of up to 20ms (separate random values are generated for each recipient of a broadcast message). The message delivery behaviour is thus highly realistic in the sense that the delivery order of messages is not guaranteed to be the same as the sent-order, and can differ from recipient to recipient.

Nodes' system-level behaviour is executed independently of their cluster-level behaviour. A node can be simultaneously idle at the system level, and master at the cluster level, for example.

Clusters are disbanded once their task has been executed to completion. Detection of task completion is the responsibility of the application level, so the cluster-level simulation-model uses a simple random distribution to model task duration.

7. PERFORMANCE EVALUATION

A simulation model of the cluster management system has been developed and is available for evaluation purposes at: http://staffweb.cms.gre.ac.uk/~ar26/Research/CurrentResearch/EmergentClusterManagement/EmergentClusterManagementMenu.htm

Using the model, robustness and scalability are evaluated by stress testing to determine the sensitivity to message loss and node failure of both the cluster-layer and the system-layer. The results presented relate to the system-layer but can be extrapolated to the cluster layer because, in terms of the maintenance of a coordinator, the two layers use fundamentally the same algorithm.

In modern networks the probability of a message being lost (or corrupted) is generally very low but is dependent on the type of network, type of medium and congestion levels in networks.

Message loss is expected to lead to false elections, as the probability of slave nodes incorrectly detecting the failure of the master is related to the probability of message loss. The simulation model is capable of randomly dropping messages. The probability of dropping messages is governed by a user-supplied parameter. The range of message-loss levels used in the evaluation include extremely high levels that are not expected to occur in realistic deployment scenarios but are included to stress-test the algorithm.

The model was configured as follows. Simulation period: 1 hour; System size: 200 nodes; Initial node state: all idle; Lower slave-pool size threshold: 2; Upper slave-pool size threshold: 4; Node failure: not active. A series of experiments were conducted, varying the message loss probability each time. The results presented are averaged over ten simulation runs per configuration.

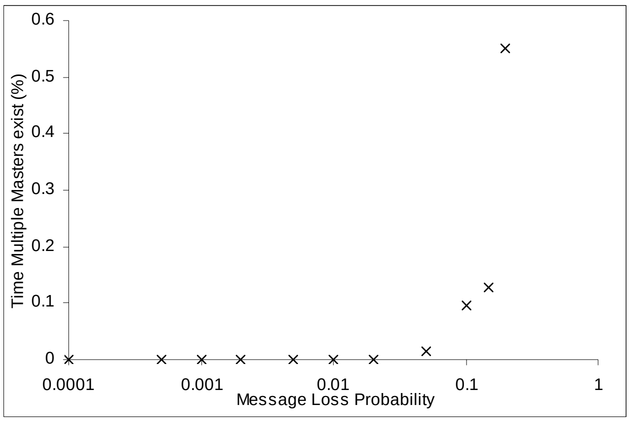


Figure 6. Sensitivity to message loss.

Figure 6 shows that the algorithm maintains a single coordinator up to a message loss level of as high as 5%. As message loss increases beyond this point the number of occurrences of multiple coordinators rises more steeply. Even so, at pessimistic message loss levels of 10%, multiple coordinators only exist approximately 0.1 % of the time and at loss levels as high as 20% multiple masters only occur for 0.552 % of the time.

In the presence of node failures, the proportion of time a coordinator is maintained is a reflection of the algorithm's ability to quickly detect coordinator failure and elect a replacement.

The simulation model assigns a MTBF value to each node, determined by a probability distribution which is governed by a user-supplied parameter.

MTBF values for computer hardware is typically of the order of thousands of hours, but when software, user-behaviour and network connectivity are taken into account, the actual MTBF value for a processing node can be much lower. In this evaluation, node failure is taken to include unexpected user-initiated reboots, hardware failure, operating-system crashes and isolation from the network. MTBF values indicate the probability of each individual node failing completely independently of, and possibly concurrently with, any other node. Node failure is expected to lead to periods when the system is leaderless.

The model was configured as follows. Simulation period: 24 hours. System size: 200 nodes; Initial node state: all idle; Lower slave-pool size threshold: 2; Upper slave-pool size threshold: 4; Message-loss probability: 0.0001; Node MTTR: 30 minutes. A series of experiments were conducted varying the node MTBF value each time. The results presented are averaged over ten simulation runs per configuration.

Figure 7 shows the effect of reducing per-node MTBF on the proportion of time the system is leaderless. Each time a coordinator node fails there is a short period, the duration of an election, in which no coordinator exists. With per-node MTBF as low as 1000 and 60 minutes the algorithm maintained a coordinator approximately 99.98 % and 99.22% of the time respectively.

The MTTR value does not directly affect these results because newly-recovered nodes always rejoin the algorithm in the idle state at the system level, and the not-in-a-cluster state at the cluster level. This design approach enhances stability since a failed coordinator, on recovery does not try to regain its previous status.

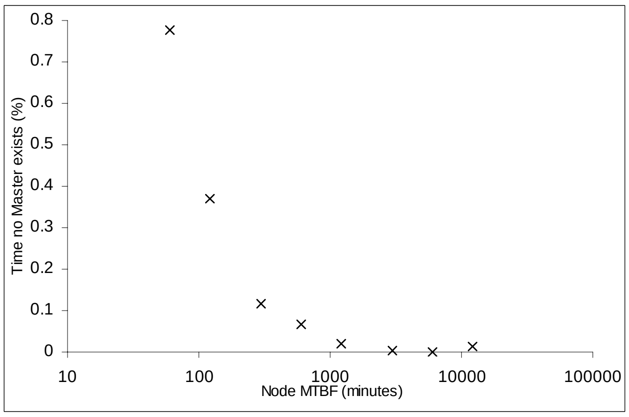


Figure 7. Sensitivity to node failure.

The effect of the system-level being leaderless is that clusters temporarily cannot be created. This can be dealt with at the application-level by arranging that unanswered cluster creation requests are re-submitted after a short delay.

The main impact of the loss of leader at the cluster-level is that task execution and results-collection will not be coordinated. To overcome this problem, the small amount of state held by the cluster coordinator can be appended to its periodic master message broadcasts for caching by the other members of the cluster. Any node that subsequently takes over the coordinator role then has access to the state. The refined details of this aspect will be dealt with in follow-up work.

The model is used to evaluate cluster-layer efficiency in terms of the relationships between the number of messages required to create clusters, cluster size, and system size. To create a cluster, the system-level coordinator sends an *InviteMembershipBids* message to elicit membership bids. When creating relatively small clusters in large systems, it is possible that a much larger number of nodes will respond with *MembershipBid* messages than are needed to form the cluster. This would be inefficient and impact on scalability.

To resolve the issue, a delayed-bid mechanism is deployed. Nodes that are available to join an advertised cluster wait a short random period before sending their *MembershipBid* message. This spreads out, in time, the arrival of bids at the system coordinator. Once the system coordinator has received sufficient membership bids to build the cluster, it sends a

StopBids broadcast which prevents outstanding *MembershipBid* transmissions from occurring.

Consider a system of 100 nodes in which 80 are not currently members of clusters and thus available. Without the delayed bid mechanism, an *InviteMembershipBids* message to create a cluster of 10 nodes would receive 80 replies. The coordinator would then send 10 *Accept* messages. The total number of messages to create the cluster would be 91. However, with the delayed bid mechanism in place, the *StopBids* transmission will prevent up to 70 of the *MembershipBid* messages, for a cost of one additional broadcast. The random delay used is between 0 and 500ms. Thus the worst-case effect of the mechanism is to add 500ms to the cluster-creation latency. Probabilistically, for large systems, enough messages arrive at the coordinator to create typical-sized clusters before the majority of messages have been sent, allowing the cancellation message to be highly effective and limiting the effect that the delay mechanism has on the latency of cluster creation. Figure 8 shows the relationship between system size and the actual numbers of messages required to build clusters of several sizes, based on experimentation with the simulation model.

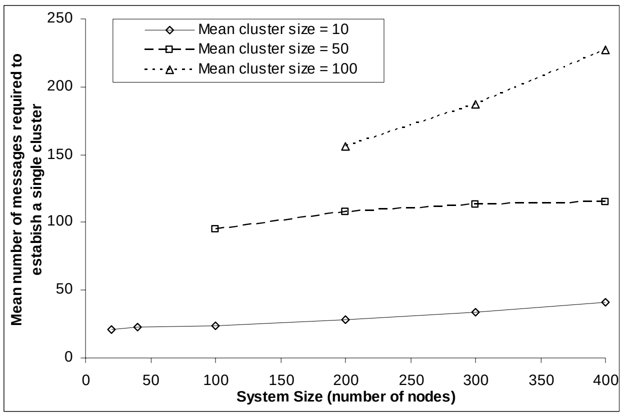


Figure 8. Mean message costs of cluster creation.

Typical savings (in terms of the percentage of potentially-generated messages) achieved by using the delayed-bid mechanism are illustrated in table 1. For example, if all messages were sent (i.e. not using the delayed-bid mechanism) then to create a cluster of 10 nodes in a 400-node system, 411 messages would be generated. However, by using the mechanism, typically 90% of these messages are avoided, approximately 41 being sent. The minimum number of messages needed is actually 21.

Table 1. Message reduction achieved by delayed bid mechanism

Cluster size (mean number of nodes)	System size (number of nodes)			
	100	200	300	400
10	78%	87%	89%	90%
50	37%	57%	68%	75%
100	-	48%	53%	55%

The system-level communication complexity for normal-mode operation and during elections have been discussed in section 4. The same election algorithm as used at the system-level operates within each cluster. Thus, in addition to cluster creation communication, the mean communication costs for the normal-mode are $O((j + 1)\ (s/10 + 1/5))$ messages per second, where j is the mean number of clusters coexisting. The election communication complexity is the same for a cluster as it is for system-level elections (i.e. very low and independent of cluster size if the cluster contains more than 4 nodes). Where clusters are sufficiently long-lived, the creation costs are amortized over time and become insignificant. The application domain implies that generally this condition will be upheld. Scalability is ensured since the normal-mode and election communication complexity are independent of system size and cluster size.

8. CONCLUSION AND FURTHER WORK

This paper proposes that self-healing should be embedded at all layers of autonomic systems, including the software platforms and core services on which they depend.

To illustrate the flexibility of this approach, a layered resource-management architecture was presented. Inspired by biological systems, the architecture embeds self-management in services deployed at each of its three layers. Each layer has the ability to self-heal and self-stabilise. This is precisely the type of design that should be considered when building higher-level distributed applications which must themselves be scalable, robust, stable and self-healing.

The main discussion in this paper concerns the design of the middle (cluster-management) layer. The design of the cluster management system incorporates a novel mix of deterministic and non-deterministic behaviour. The non-deterministic aspects of the design were inspired by interaction protocols found in natural emergent systems. These aspects impart highly robust and scalable operation, whilst the design remains simple and has a very-low-complexity communication model. The overall operation, at both the system level and the cluster level, is deterministic and stable.

Wrapping the non-deterministic core behaviour (the idle-slave interaction) to ensure that the externally-visible behaviour is deterministic required some additional communication and an increase in internal complexity (in the form of the candidate state and its accompanying transmissions and timers), but has been achieved without eroding the benefits gained.

The failure-sensitivity testing results show that under realistic operating conditions the system remains completely stable in a 200 node system. The independent variable ranges used in these experiments included extremely pessimistic values to stress-test the algorithm. The algorithm tolerates message loss levels of 20-30%. At such levels the momentary occurrence of multiple coordinators is in the order of 0.5 – 1% of operational time. The algorithm adequately tolerated realistic per-node MTBF values, quickly returning to a legal state after master failure.

A main contribution of this paper has been to demonstrate that ‘engineered emergence’ can be a powerful design paradigm for high-quality distributed applications which are highly efficient, scalable and robust, yet have quite simple internal behaviour.

The investigation of emergence-inspired computing as a foundation for the development of autonomic systems is ongoing.

The application-layer is being extended to support a wide-range of autonomic applications including highly-adaptive self-load-balancing parallel applications [24].

Strategies for generalising the effective embedding of self-managing and self-healing behaviour into many levels throughout a system are being explored.

9. REFERENCES

[1] F. Maturana, P. Tichy, P. Slechta, F. Discenzo, R. Staron and K. Hall, "Distributed Multi-Agent Architecture for Automation Systems", **Expert Systems with Applications**, Vol. 26, No. 1, Elsevier Science, 2004, pp.49-56.

[2] K. Shindo, N. Koshizuka and K. Sakamura, "Large-Scale Ubiquitous Information System for Digital Museum", **Proceedings of 21st International Conference on Applied Informatics**, IASTED, Innsbruck, February 2003, pp. 172-178.

[3] J. Koehler, C Giblin, D. Gantenbein and R. Hauser, **On Autonomic Computing Architectures**, Research Report (Computer Science) RZ 3487(#99302), IBM Research (Zurich), 2003.

[4] N. Shadbolt, "Nature-Inspired Computing", **Intelligent Systems**, IEEE, 2004, pp.2-3.

[5] S. Johnson, **Emergence: The connected lives of Ants, Brains, Cities and Software**, Penguin Press, London, 2001).

[6] R. Anthony, "Transparency-Driven Design of Distributed Software", **Proceedings of 19th International Conference on Applied Informatics**, IASTED, Innsbruck, February 2001, pp. 45-50.

[7] G. Eschelbeck, "Parallel computation with dynamic load distribution for locally distributed NT environments", **Journal of Microcomputer Applications**, Academic Press, Vol. 18, 1995, pp. 193-201.

[8] P. Krueger and R. Chawla, "The Stealth Distributed Scheduler", **Proceedings of 11th International Conference on Distributed Computing Systems**, IEEE, 1991, pp. 336-343.

[9] R. Anthony, **Load-Sharing in Loosely-Coupled Distributed Systems: A rich-information approach**, D.Phil. Thesis, Dept. Computer Science, University of York, UK, March 2000.

[10] A. Goscinski, "Finding, expressing and managing parallelism in programs executed on clusters of workstations", **Computer Communications**, Vol. 22, Elsevier Science B.V., 1999, pp. 998-1016.

[11] C. Soleimany and S. Dandamudi, "Performance of a distributed architecture for query processing on workstation clusters", **Future Generation Computer Systems**, Vol. 19, Elsevier Science B.V., 2003, pp. 463-478.

[12] A. Goscinski, "Towards an operating system managing parallelism of computing on clusters", **Future Generation Computer Systems**, Vol. 17, Elsevier Science B.V., 2000, pp. 293-314.

[13] J. Jaen-Martinez, "The Java Management Extensions (JMX): Is Your Cluster Ready for Evolution?", **Journal of Parallel and Distributed Computing**, Vol. 60, Academic Press, 2000, pp. 1341-1353.

[14] L. Lundberg and C. Svahnberg, "Optimal Recovery Schemes for High-Availability Cluster and Distributed Computing", **Journal of Parallel and Distributed Computing**, Vol. 61, Elsevier Science, 2001, pp. 1680-1691.

[15] H. Unger and G. Hipper, "LYDIA – Load Sharing for PVM-applications in a Workstation Cluster", **Proceedings of 9th International Conference on Parallel and Distributed Computing and Systems**, IASTED, Washington DC, October 1997, pp. 183-187.

[16] F. Turck, S. Vanhastel, B. Volckaert and P. Demeester, "A generic middleware-based platform for scalable cluster computing", **Future Generation Computer Systems**, Vol. 18, Elsevier Science B.V., 2002, pp. 549-560.

[17] R. Brightwell, L. Fisk, D. Greenberg, T. Hudson, M. Levenhagen, A. Maccabe and R. Riesen, "Massively parallel computing using commodity components", **Parallel Computing**, Vol. 26, Elsevier Science B.V., 2000, pp. 243-266.

[18] T. Priol, G. Alleon, "A client/server approach for HPC applications within a networking environment", **Future Generation Computer Systems**, Vol. 17, Elsevier Science B.V., 2001, pp. 813-822.

[19] J.L. Casti, **Complexification: Explaining a Paradoxical World Through the Science of Surprise**, Abacus, London, 1994.

[20] R. Genet, **The Chimpanzees who would be Ants: A Unified Scientific Story of Humanity**, Nova Science Publishers Inc, New York, 1997.

[21] M. Gell-Mann, **The Quark and the Jaguar: Adventures in the Simple and the Complex**, Abacus, London, 1994.

[22] J. Ferber, **Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence**, Addison Wesley Longman, Harlow, England, 1999.

[23] Anthony. R, Emergence: A Paradigm for Robust and Scalable Distributed Applications, **1st Intl. Conf. Autonomic Computing (ICAC)**, IEEE, New York, 2004, pp.132-139.

[24] R. Anthony, "Self-Configuration in Parallel Processing", **3rd International Workshop on Self-Adaptable and Autonomic Computing Systems - SAACS '05 (DEXA 2005)**, IEEE, Copenhagen, Denmark, August 2005, 175-180.