Towards A General Model of Organisational Adaptation: Pipelines

Matthew Shaw, Jeroen Keppens, Michael Luck, and Simon Miles

Department of Informatics, King's College London, Strand, London
{matthew.shaw}@kcl.ac.uk

Abstract. Large scale information systems are increasingly structured around flexible workflows of services providing a range of functionalities that are configured to suit particular needs, yet this flexibility can bring a lack of organisation in the ways in which services are combined. Particular system structures bring different benefits to an application in terms of efficacy and efficiency but sometimes need to reorganise as their circumstances change. In this context, this paper seeks to establish techniques for reorganisation that preserve particular topologies in support of their recognised benefit for the target applications. The contributions are twofold: first, a general vision of reorganisation of defined structures, in which structure is preserved but efficient and efficacy if optimised; and second, a specific solution for the case of pipelines, reorganising to optimise key application-specific metrics, while preserving structure. The paper is thus the starting point for a more ambitious general programme of research.

Keywords: Self-organisation, reorganisation, pipelines

1 Introduction

Large scale information systems are increasingly structured around workflows of services providing a range of functionalities that are configured to suit particular needs. Indeed, the construction of combinations of services to form systems satisfying application-specific demands offers a flexibility that is often missing in rigid system structures. However, this flexibility can bring a lack of organisation in the ways in which services are combined. For example, many applications are naturally hierarchal in nature and are thus best suited to a hierarchical organisation of services. Other applications will suggest alternative structural arrangements, some of which may correspond to well-known organisations, and some of which may be ad hoc instead.

Now, the suitability of an organisational structure to an application in this sense lies in the efficacy and efficiency of the structure in supporting the particular goals of the system. This may be in terms of minimising load, in maximising throughput, or in other such objectives to be optimised. This paper recognises the value of such organisational structures yet seeks to provide a means of enabling them to reorganise as their circumstances change. Indeed reorganisation is known to be a valuable technique in the armoury of modern computing systems, motivated in part by increasing interest in areas such as autonomic computing. Importantly, however, rather than allow arbitrary

ad hoc structures to emerge, this paper takes a different standpoint in seeking to establish techniques for reorganisation that preserve particular topologies in support of their recognised benefit for the target applications.

The contributions of this paper are twofold. First, the paper presents a general vision of reorganisation of defined structures, in which structure is preserved but efficient and efficacy optimised. Ultimately, this will lead to a library of techniques for structure-preserving reorganisation across different topologies, and potentially a means to transform between topologies as needs demand. Second, the paper instantiates this broad vision with a specific solution for the case of pipelines, reorganising to optimise key application-specific metrics, while preserving structure. It is thus the starting point for the more ambitious general programme of research.

The paper is structured as follows. The next section motivates the paper as whole by presenting a motivating scenario and a task allocation model setting out the problem we address. Then, in Section 3, we describe the main contribution of the paper, the techniques for reorganising, both in general and in the specific case of pipelines. Section 4 presents the initial results obtained with our techniques, before reviewing related work in Section 5, and concluding in Section 6.

2 Task Allocation and Execution Model

2.1 eScience Scenario

To motivate our work, we introduce a scenario based in the domain of eScience. Consider a large, potentially global network of electronic resources (such as devices, or even data) that are owned by different institutions, all willing to pool their individual resources in order to gain access to a larger set of shared resources that would otherwise be unavailable to them. All resources are networked, with some performing computationally intensive tasks like the analysis of large scientific data sets such as resulting from the Large Hadron Collider (LHC) at CERN [5, 6]. Processing this data takes considerable time, and we want to process it as quickly as possible. If there is only one task that cannot be processed concurrently across multiple machines, then the optimal solution is for it to be processed on the fastest machine.

Now, suppose there are two research centres, each using a *particle accelerator* (PA), and a *laser research apparatus* (LRA) respectively, and both generate large data sets. Such data needs to be stored and then processed, but neither research facility has the ability to do so. However, there are two data storage units, (DSU1 and DSU2), that are each capable of storing 100 units of data, and two supercomputers, (SC1 and SC2), that are capable of processing data, with SC1 processing data faster than SC2. This is summarised in Table 1.[1]

If PA and LRA simultaneously perform experiments, respectively producing 50 units and 120 units of data, PA can store its data at DSU1, but LRA stores 100 units at DSU2 and 20 units at DSU1, since neither DSU has the capacity for 120 units. PA's data is then passed to the fastest supercomputer, SC1, for processing and, since SC1 is now busy, LRA's data is passed to SC2 despite it being slower. This is illustrated in Figure 1.

[1] The capabilities of PA and LRA are not relevant, so are omitted.

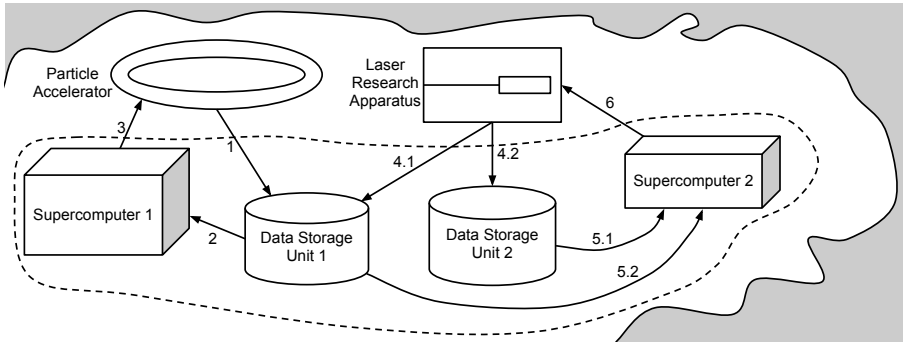


Fig. 1. Visualisation of the eScience scenario

		Computational Devices					
		SC1	SC2	DSU1	DSU2	PA	LRA
Services	Data Storage	N	N	Y	Y	N	N
	Processing Power	Y	Y	N	N	N	N
Hardware Specs	Memory	-	-	100	100	-	-
	Processing Power	Faster	Slower	-	-	-	-

Table 1. The capabilities and resources available in the eScience scenario

2.2 Task Model

In this scenario we require the different computational entities to undertake various tasks (storing data or processing it), and to pass these tasks to others if the entities themselves cannot execute them. The key *task* of this scenario is to analyse data. In this sense, a task satisfies a particular *requirement*, where that requirement amounts to a specification of the *services* needed to perform that task, and the time for which each such service is needed. Tasks may also be decomposed into subtasks (potentially with ordering constraints): for example, to analyse data it must first be stored and then processed.

In our example, DSU1 and SC1 offer *services* for processing and storing data. The details of such services are unimportant for our purposes, and we simply specify the set of all services, $S = \{s_1, s_2, \dots\}$. Clearly, in order to fulfil tasks, services must perform some work. A *requirement* is a specification of the services and the amount of work needed from each in order to achieve the task. For simplicity, we assume that all services provide the same amount of work, or effort, per unit of time, and we use time as a simple proxy for an amount of work. In this way, a service may be required for 3 units of time, while another is required for 6 units of time. A requirement r thus takes the form $(s, reqt)$, where $s \in S$ is the required service, and $reqt \in \mathbb{Z}+$ is the amount of time for which it is required. The enactment of a service to satisfy a task's requirement is encapsulated as a *service instance*, in the form, $si = (t, s)$, where s is the service satisfying task t. Since tasks are decomposable, as indicated above, they are represented in a tree structure, as in Figure 2, where each vertex indicates a subtask, and each line an ordering constraint between subtasks. These constraints mean that t_1 must be completed before t_2 and t_3 can begin, but once t_1 is completed, t_2 and t_3 can

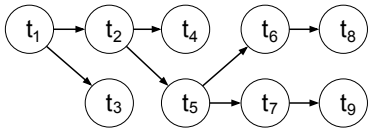


Fig. 2. Example of a task hierarchy.

be executed concurrently. To represent this additional complexity, a task takes the form $t = (r, \text{SUBT})$ where *r* is a requirement, and SUBT is a set of subtasks.

Given all this, devices such as PA or DSU1 are represented as *agents*, that use their services SERV to execute tasks that are in its list of tasks, TASKS. Each agent has a capacity $cap \in \mathbb{Z}+$ limiting the number of service instances they can run concurrently, which represents a limit of resources such as memory in a data storage unit. As each service is used, a service instance is created and added to a's set of current service instances SI, such that $|\text{SI}| \leq cap$. Once all of a task's requirements have been met, the service instance will be removed, allowing more to be created. Finally, an agent has a set CON of connections with other agents. An agent a, therefore, is represented as $a = (\text{SERV}, \text{TASKS}, cap, \text{SI}, \text{CONN})$.

2.3 Task Allocation

Given our basic model above, we can now consider how agents are allocated tasks. We adopt a simple model with the assumption that time is in discrete units, with a number of rounds, in each of which an agent a undertakes two major activities: it manages its TASKS list; and it executes tasks. In managing its tasks, a first places any received tasks in its TASKS list. Then, in order of arrival, each task *t* in the list is reviewed: if *t*'s requirements can be satisfied directly by a, and a has capacity to do so, a creates a service instance *si* to execute *t*, adds *si* to SI, and removes *t* from the TASKS list; if a can satisfy *t*'s requirements, but does not currently have capacity to do so, then the task remains on the list, waiting for capacity to become available; finally, if *a* cannot satisfy *t*'s requirements, then it must find another agent to which to delegate the task.

Once *a* has finished managing its TASKS list for the current round, it begins executing tasks, each of which is represented as a service instance in SI. As indicated above, tasks require a service for a specified number of rounds (*reqt* in the task's requirement), so a service instance persists until *reqt* has elapsed, at which point the service instance is removed. If a completed task has subtasks, each subtask is added to the TASKS list so that they can be allocated or executed on the next round. In our model, each agent *connects* to a set of other agents, which are the only agents with which it can communicate, giving an organisational structure. Then, if an agent wants to find a service to which to allocate tasks, it performs a depth first search across the organisational structure until an appropriate service is found. Clearly the links between agents determine how easily an agent can find another agent offering a required service.

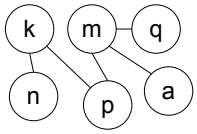


Fig. 3. An example of an organisational structure represented as a graph.

2.4 System Metrics

While any system can be designed to achieve different goals, in this paper we assume the most obvious goal of executing tasks as quickly as possible. In this subsection, therefore, we complete the description of our model by introducing the ways in which we are able to measure the performance of our system, first through three main metrics: *load*, *throughput*, and *messages*, and subsequently through ways of measuring other relevant systems properties.

The *load* of an agent is specified as $(|\text{SI}|/cap) \times 100$, where $|\text{SI}|$ is the number of service instances. The result is a percentage, indicating the degree of usage from full capacity at 100% to complete idleness at 0%. An agent is overloaded when its load is 100% and there are tasks in its TASKS list for which it satisfies the requirements. An agent's *throughput* is the number of tasks it *finishes* executing each round. The system's throughput is the total number of tasks that finish being executed at each time step. This value is suboptimal if tasks are waiting to be executed by one overloaded agent, but there are other agents that can execute these tasks immediately instead. Finally, when trying to locate a service to allocate a task that an agent cannot perform itself, the agent sends query messages to others for the required service. Similarly, when a task is actually allocated, another message is sent by message. If the organisational structure is well designed, then agents will be close to the required services, thus sending fewer messages.

In addition to these three main metrics for performance, we can consider others that may be useful in what follows. For example, an agent's *task arrival rate* is the number of tasks that it receives each round, its *executable task arrival rate* is the number of tasks that it is capable of executing it receives each round, and its *allocation task arrival rate* is the number of tasks that it cannot execute it receives each round, and so must allocate elsewhere. We can also consider how frequently each service *type* is used, as *service frequency*.

3 Adapting Organisational Structure

The model just described specifies how tasks may be allocated and executed in distributed systems exemplified by our eScience scenario. However, since the structural organisation of the system is determined by the connections between agents, which in turn constrain the way in which tasks are allocated, we may not have an efficient system, or not as efficient as may be possible. For example, load (tasks requiring allocation and execution) may be poorly distributed across the agents (and their services) as a result. In consequence, this section describes a set of techniques that allow a system to be

reorganised in such a way as to improve its efficiency in line with the metrics described above.

Importantly, we want techniques that adapt a system so that efficiency is improved while also preserving the organisational structure the system is designed to exploit. These techniques might apply to pipelines, hierarchies or other structures, each of which has particular characteristics. In this respect we seek to provide a general form of reorganisation, a template that can be instantiated for different structures, while at the same time drawing them all together in a coherent whole. In this paper we therefore discuss aspects of the general tools, but focus in particular on the case of simple pipelines.

In what follows, each reorganisation technique is designed around a two-stage process, where the first stage is concerned with analysing the organisational structure and determining what links between agents should be added to or removed, while the second stage ensures that the adaptation (the addition or removal of links) preserves the particular structure or topology. In this paper, due to space constraints, we illustrate the general concept by means of one of the simplest topologies, *pipelines*, but seek to apply this template for reorganisation to other structures, in the same way, proposing changes to the links between agents, while preserving the overarching structure.

3.1 Analysing Structure

The first stage in reorganisation thus requires an analysis of an organisational structure in order to populate a *change set*, *C*, in which each element $c = (a_1, a_2, action) \in C$ indicates a change to be made, where a_1 and a_2 are agents that are either connected, or have the potential to be connected, and *action* is an element in the set $\{create, remove\}$. Now, since an organisational structure is, in essence, a graph defined by the *agents* (or vertices) involved and the *connections* (or edges) between them, the initial analysis uses graph metrics to find potential problems in the organisational structure, as follows.

By convention, a vertex is denoted by *v* such that $v \in \mathrm{V}$ where V is the set of all vertices, and an edge is denoted by e such that $e \in \mathrm{E}$ where E is a set of all edges. The end points of an edge are always two vertices, so an edge *e* is a vertex pair $e = (v_1, v_2)$. We thus represent a graph as $\mathrm{G} = (\mathrm{V}, \mathrm{E})$ [8]. Similarly, we denote a multiagent system $\mathrm{MAS} = (\mathrm{A}, \mathrm{C})$, where A is the set of all agents and C is the set of all connections between the agents in A. In a graph $\mathrm{G} = (\mathrm{V}, \mathrm{E})$, a path is a set of ordered vertices $P = \{v_1, v_2, \dots\}$ such that each vertex is connected to the next, in order, and the same vertex does not appear twice. A path is a traversal of a graph.

Given this description, we can introduce some standard graph metrics that may be used to undertake our initial analysis. First, the *degree of connectivity*, or the *degree*, of a vertex v is the number of edges that v is a part of, and is denoted by $deg(v)$. Second, the length of a path is the number of edges that it traverses. The *shortest path* is the path with the least number of edges, and the length of the shortest path between v_1 and v_2 is the *distance* between v_1 and v_2, denoted by $d(v_1, v_2)$.

Now, one of the key properties of an agent is its ability to interact with others. In this context, Freeman reviews different concepts of *centrality* [2], which is accepted as playing a significant role in influence in social networks and on the efficiency of group behaviour. For example, in Figure 3, an intuitive assessment suggests that *m* and

p are central. According to Freeman, this is for three reasons: first, *m* has direct contact with the largest proportion of the system; second, *m* is closest to all other nodes in the system; and third, *m* and *p* are in control of much information that may pass between nodes [2]. Different measures of centrality can be defined: *degree centrality*, *closeness centrality*, and *betweenness centrality*.

Degree Centrality states an agent's centrality based on its degree, compared to the degree of all other agents. The higher the value the more central the agent. The degree centrality of an agent *m* is $\mathrm{C}_d(m) = deg(m)$.

Betweenness Centrality is concerned with the position of an agent in relation to others. An agent *m* is *between* a pair of agents *n* and *q* if it appears in the shortest path connecting *n* and *q*. Betweenness centrality is the number of agent pairs that *m* appears between: the higher the value, the more central. In Figure 3, agent *m* is between seven agent pairs, while agent *p* is between six pairs, so agent *m* is the most central. Calculating *m*'s betweenness centrality is more complex when there is more than one shortest path between two agents, because *m* will no longer be between the two agents all of the time. If there are two shortest paths between *n* and *q*, and *m* appears in only one, then there is 0.5 chance that a message between *n* and *q* will pass through *m*. More formally, *m*'s betweenness centrality can be measured by the number of shortest paths it appears in $\sigma_{nq}(m)$, divided by the total number of shortest paths σ_{nq}, as in Equation 1.

$$\mathrm{C}_\mathrm{B}(m) = \sum_{m \neq n \neq q \in \mathrm{A}} \frac{\sigma_{nq}(m)}{\sigma_{nq}} \tag{1}$$

Closeness Centrality gives an agent *m*'s centrality based on how close it is to the rest of the system. We can use Dijkstra's algorithm to find the shortest spanning tree rooted at agent *m*, consisting of the shortest path from *m* to all other agents. Agent *m*'s closeness centrality is the the sum of the length of all of these shortest paths as in Equation 2; the lower, the value the more central.

$$\mathrm{C}_\mathrm{C}(m) = \sum_{n \in \mathrm{A} \setminus m} d(m, n) \tag{2}$$

3.2 Proposed Changes to Structure

The above metrics provide a means of understanding certain properties of our organisational structures so that we are able to consider the changes that should be made. In the previous discussion, we have considered metrics in general, and continue this general analysis in this section in which we consider two simple ways of modifying a structure to give greater efficiency.

First, we seek to reduce the *distance* between an agent and the service it uses most frequently. Here, for each agent *m*, a depth-first search is used, starting at that agent, and searching through the entire organisational structure to find the closest instance of the service *s* that *m* most frequently uses, but that *m* and its neighbours do not offer. If the closest agent *n* offering service *s* is more than a specified number of hops away from agent *m*, then an element is added to the change set recommending a direct connection between *m* and *n*: $C = C \cup \{(m, n, create)\}$.

Second, we seek to decrease the load on overloaded agents by decreasing their closeness centrality, and moving them away from the centre of the system. Here, each agent's load is measured. If at least one agent has a load of 100%, with tasks waiting in its TASKS list that it has the capability to execute, then the system contains at least one overloaded agent. In response, each agent's centrality value is calculated and overloaded agents are moved one step away from the centre of the system, as follows. An overloaded agent *m* finds the neighbour *n* with the highest centrality, and the neighbour *p* with the lowest centrality. It also finds *q*, a neighbour of *p* with the lowest centrality out of *p*'s neighbours. A new connection is then created between *m* and *q*, while the connection between *m* and *n* is removed. The resulting elements in the change set are: $C = C \cup \{(m, q, create), (m, n, remove)\}$.

3.3 Preserving Structure

To this point, the techniques introduced, and indeed the methodology for doing so, have been presented in the most general way. However, this last part of the process requires us now to instantiate our work with the specific organisational structure or topology that constrains the changes proposed. In fact, as indicated in the introduction to the paper, the aim of our work is to facilitate reorganisation by building up a library of techniques and constraints in a stepwise fashion across different topologies, and then to generalise these instances to provide a generic model. This paper describes the early stages of this programme of work, and is restricted, as a first step, to *pipelines*, one of the simplest possible topologies, in order to illustrate the general approach. Subsequent work will consider application to hierarchies, matrices and other structures, but that is not considered in this paper.

In a pipeline, all of the vertices are lined up sequentially. Each vertex has a degree of connectivity of exactly 2, except for the vertices at the ends of the pipeline, which have exactly 1. In a multi-agent system, this translates to each agent having a maximum of two connections. If we wish to reorganise a pipeline, then we cannot change the number of connections, but we can change the connections themselves as long as we preserve the pipeline properties. The only legal change to such a structure is thus the positioning of each agent. Note that to do this we have only two possible changes from the change set: remove a connection and create (or add) a connection. However, if we remove a connection from a pipeline, we cannot preserve the structure since it breaks the pipeline irretrievably. For this reason, we do not entertain this possibility for pipelines (though we will do so in future work for other topologies), and focus here only on creating connections (though, confusingly, we will see that this will require removal of connections as part of the process of structure preservation).

To illustrate, suppose we have a pipeline with five agents, as shown in Figure 4 part 1, and a change set in which the first element states that a connection should be created between agents *p* and *n*. If this connection is created, then the organisational structure is no longer a pipeline. To ensure that the pipeline is maintained, the following changes must also be added to the change set: remove connections between *m* and *p*, *q* and *p*, and *k* and *n*, and create connections between *m* and *q*, and *k* and *p*. With all these changes, a connection can be created between *p* and *n*, while maintaining a pipeline. In fact, this is one way to achieve the result we aim for, but the same can also be achieved by adding

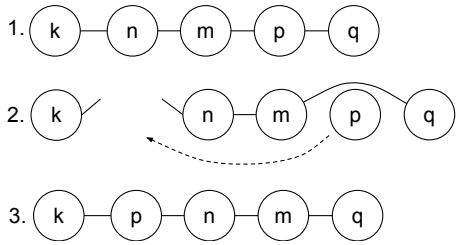


Fig. 4. An example of enacting a change in a pipeline.

the connections elsewhere. For ease of exposition, we will not consider the alternatives in this paper, but note that they lead to the same outcome.

More formally, the change set is altered as follows: for each $(x, y, create) \in C$, x is moved next to y. As just noted, this is possible in different ways; here we do so by removing agent x, then reinsert x next to y in just one way.

Remove Agent Removing x from a pipeline depends on x's position in the pipeline. If x has one connection then it is at the end of the pipeline, and so its only connection is removed. If x has two connections then it is in the middle of the pipeline, so both of its connections are removed. Then, a connection is created between x's previous neighbours, ensuring that the pipeline is maintained. In both instances, the result is a pipeline, excluding the single disconnected agent x.

Reinsert Agent Reinserting x next to y similarly depends on y's position. If y has one connection then it is at the end of the pipeline, so a connection is created between x and y, and x is now at the end of the pipeline. If y has two connections then it is in the middle of the pipeline, so we must decide on which side of y to reinsert x. To do so we randomly select one of y's neighbours, z, remove the connection between y and z, and create connections between x and y, and x and z. In both instances, the result is a pipeline where x is directly connected to y.

As indicated above, we do not consider the removal of connections due to the constraints of a pipeline, since this cannot be done without irretrievably breaking the structure, and these changes are therefore simply eliminated from the change set.

4 Evaluation

Given these techniques designed to reorganise pipelines to increase the performance of a system, we undertook a series of experiments to show their impact and indeed whether they are successful. In order to provide a baseline for comparison, we experimented with *random* changes to the structure as well as the *reducing distance* and *decreasing load* changes.

As described above, the changes involved in pipelines are to move agents from one position in a pipeline to another. Whenever reorganisation is triggered in the random approach, 10 agents are randomly selected and moved next to another randomly selected

agent that it is not currently connected to. Moving an agent a to an agent b consists of: removing the connection between a and each of its neighbours; creating a connection between each of a's old neighbours; removing the connection between b and one of its neighbours; and creating connections between a and b, and between a and b's old neighbour.

In what follows, we describe our results from simulating the task allocation and execution model described above. The simulation consisted of 100 agents in a pipeline with a random initial structure, where the agents receive tasks, and allocate or execute them. The simulation consisted of 2000 rounds, in each of which a number of tasks are randomly generated, and allocated to agents at random, regardless of the services offered by agents. The number of tasks generated at each time step varies according to the Poisson distribution with a mean task arrival rate of 10. On each round, agents follow the behaviour described in Section 2.3, after which reorganisation is potentially triggered: in rounds 0–499, there is no reorganisation, but in rounds 500–2000 reorganisation is triggered each time. Each simulation was repeated 15 times, with all results being averaged over these multiple runs.

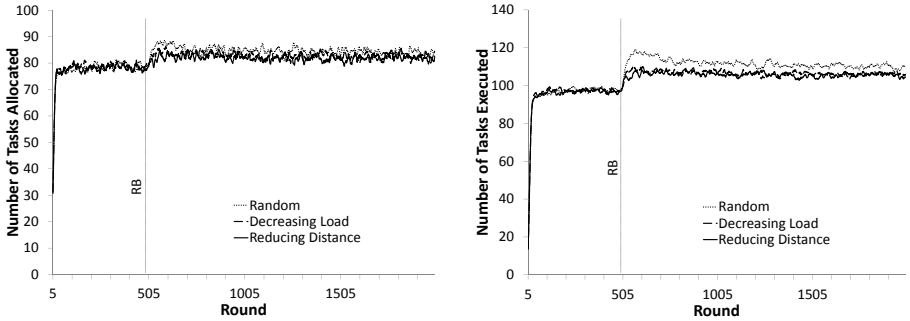
4.1 Task Allocation and Execution

Throughout the simulation, the number of tasks that each agent is allocated and executes is counted, and the mean number across all agents plotted over time. Figure 5(a) shows how many tasks are allocated on average at each time step, and moreover that there is little variation between all three techniques, which each display a small increase in the number of tasks allocated. Random reorganisation shows a slightly larger increase, but the difference is minimal.

Figure 5(b) shows how many tasks are executed on average at each time step, and again that there is little variation between the *reducing load* and *decreasing distance* techniques. However, we see better performance from random reorganisation.

In our model, locating services and sending a task between agents takes no time,[2] so the number of tasks executed increases because, rather than waiting on an overloaded agent's TASKS list, after reorganisation individual tasks are allocated to agents with spare capacity. In turn, this increases the number of subtasks being released for execution, so the number of tasks allocated also increases. The increase appears in all simulations, including *random*, indicating that the increase in task execution is due to the organisational structure as a whole, rather than any specific change instance. Since an agent uses the relevant service on the closest agent, it will always use the same agent for a particular service unless the structure changes. Globally, this means that without any change to structure, agents whose services are not initially used will never be used, while those services that are initially used will be used regularly. However, by making regular changes to the organisational structure (random, or otherwise), the closest instance of a service to an agent will also regularly change.

[2] Instead, the number of messages needed, and the distance that tasks travel, are considered separately.



(a) Mean number of tasks allocated. (b) Mean number of tasks executed.

Fig. 5. Comparison of the number of tasks allocated and executed.

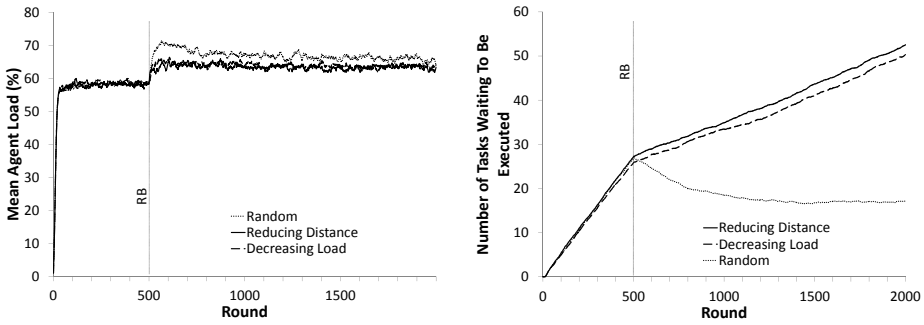
4.2 Load and Waiting Tasks

Throughout the simulation, the load of each agent, the number of tasks waiting to be executed, across all agents, are recorded. Figure 6(a) shows the average load plotted over time, indicating that though all techniques increase the load of the system, random reorganisation improves loading the most significantly. Figure 6(b) shows the number of tasks waiting to be executed, with *reducing distance* and *decreasing load* both decreasing the rate at which tasks accumulate, but increasing the number of waiting tasks. In contrast, random reorganisation executes tasks faster than they arrive, so the number of tasks waiting to be executed decreases. However, this effect begins to plateau after 1,000 time steps. Overall, each technique offers some improvement, but random reorganisation is effective enough to address the accumulation of tasks before reorganisation. It is understandable that the first simulation does not significantly improve performance, since the desire to be closer to one instance of a required service does not aid in the distribution of load.

The second simulation tries to move overloaded agents away from areas of high centrality so that overloading can be avoided, and this seems to be only partially achieved. This could be for one of two reasons: either centrality is not a good enough indicator of the potential load of each agent, so using it to determine where to move an overloaded agent is not sensible; or the reaction to finding an overloaded agent is not sufficient. In this latter case, instead of moving an overloaded agent away from areas of high centrality step by step, agents should be moved faster. Random reorganisation performs best out of the three techniques, because the changes it makes are stronger, encouraging the use of multiple instances of services by changing the organisational structure, without changing the behaviour of agents. While we do not believe that random reorganisation is the most effective, it clearly has some properties that can improve the effectiveness of a reorganisation process.

4.3 Messages Sent and Average Distance

Our model does not directly account for the time spent locating services, and instead we consider this separately by counting the number of messages sent. Every message sent



(a) Mean load of all agents. (b) Mean number of tasks waiting.

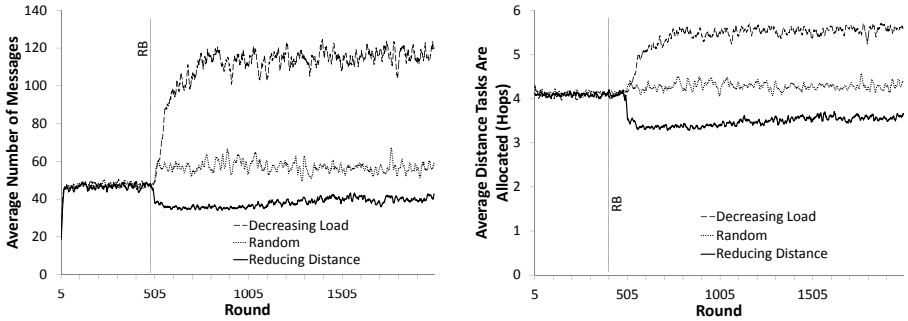
Fig. 6. Comparison of average agent load, and the number of tasks waiting.

accounts for time that a task is waiting to be executed, either because a service is being located, or because the task is being moved to an agent that can execute it. Figure 7(a) shows the number of messages sent over time, indicating that this varies greatly from technique to technique. *Decreasing load* increases the number of messages required significantly, random reorganisation requires slightly more messages, and *reducing distance* successfully decreases the number of messages substantially. Figure 7(b) shows the reason for the difference in the number of messages, in terms of the average distance tasks are moved at each time step. With *decreasing load*, the distance from required services is increased, random reorganisation has no effect on the distance from required services, and *reducing distance* successfully moves agents closer to the services they require more frequently.

Though we previously showed that random reorganisation can execute tasks faster because it most effectively distributes load, this does not take into consideration the delay caused by locating services. Here, we can see that random reorganisation has no effect on the time required to allocate tasks, whereas by actively moving agents closer to the service they use most often, tasks can be allocated more quickly, reducing the time between a task being initially provided to the system, and the time the tasks start to be executed.

4.4 Number of Changes

Figure 8 shows the number of connections changed each time step while reorganising. Before round 500, no connections are changed, but afterwards, random reorganisation makes the least number of changes, which do not vary since this is fixed and not triggered by some conditions. *Reducing distance* initially makes a massive number of changes, changing nearly 800 connections, but this almost instantly falls to 100 connections each round, and then slowly rises to a plateau. The initial spike is due to the initial structure being badly organised, but this spike quickly falls when a better structure is found. However, reorganisation does not stop completely. This is because an agent a may have a service that many other agents desire, but a single agent can only have two neighbours at most, creating competition to be directly connected to agent a.



(a) The mean number of messages sent. (b) The mean distance tasks are allocated.

Fig. 7. Comparison of the number of messages sent, and the distance tasks are allocated.

In addition, every time an agent *a* moves, it potentially triggers its neighbours to move if *a*'s neighbours relied on a service *a* offered. *Decreasing load* has no initial spike, but instead rises to a plateau, making more changes than *reducing distance*.

4.5 Summary

We can see that the performance of each technique varies according to what we want to achieve. Random reorganisation is an effective way to encourage the use of multiple service instances, which in turn distributes load. However, this does nothing to increase the time between a task initially arriving, and the time at which execution begins. Nevertheless, each of our techniques can have a positive effect on different phases of the task allocation and execution lifecycle. Potentially by combining these techniques, and

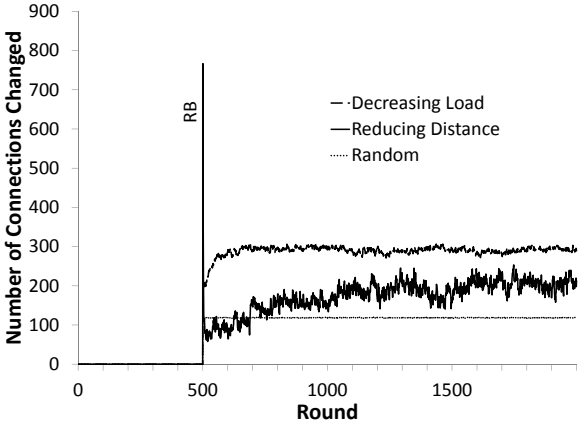


Fig. 8. The number of connections changed.

determining when to use each, we may be able to optimise reorganisation more effectively. Clearly the challenge lies in how to combine them and how to recognise when each is appropriate.

5 Related Work

Gershenson introduces reorganisation as a means of increasing the number of tasks a distributed system can execute, by decreasing communication delay arising from both transmission (latency of sending messages) and work to be performed before a reply can be sent (decision delay) [4]. In essence, this is concerned with locating the agent *a* that suffers the most from transmission and decision delays combined, the agent *b* that is a neighbour of *a* and causes the most decision delays, and the agent *c* that causes the least decision delays in the whole system. Then, the connection from *a* to *b* is removed, and a connection created from *a* to *c*. This technique was tested on a number of topologies (random-homogeneous, random-normally distributed, symmetrical, and scale-free), with results suggesting that: delay can be diminished, increasing the number of tasks executed; and the more connections, the longer to reorganise. While tackling a similar problem to this paper, Gershenson's consideration of topologies is not in their preservation but only their initial state.

Sims et al. introduce a self-organisation technique for a distributed sensor network that tracks the position and movement of vehicles [9]. Each sector involves a group of agents responsible for vehicles within the sector; problems arise when a vehicle moves along a boundary between sectors, requiring inter-sector communication with large overheads. Reorganisation here aims to minimise this communication by adjusting sector membership: if sector s_1 regularly needs information from a sensor in another sector s_2, and s_2 rarely uses it, then the sensor can be moved from s_1 to s_2, increasing global utility. This technique can adapt the organisational structure of a distributed sensor network, while maintaining the hierarchical structure in each sector. However, this is a basic two-tier hierarchy, and the changes do not consider the agent's position in the hierarchy, but rather the utility of an agent's capability (what area an agent can monitor). In contrast, our work is concerned with the organisational structure itself. Abdallah and Lesser extend the work of Sims et al. and improve the self-organisation with reinforcement learning [1]. While this improves the effectiveness of reorganisation, it still focuses on the performance of individual agents in the system, rather than on the organisational structure itself.

Gaston and Jardins introduce a couple of techniques to adapt the organisational structure of a set of agents so that teams can more easily be formed to execute tasks [3]. This is relevant to our work in providing an initial attempt at adaptation based purely on organisational structure, rather than on application-specific information. The results show that structural adaptation can be effective, and lead to better performance, but with a very high number of changes.

Similarly, Kota et al. introduce a reorganisation technique for the adaptation of problem-solving agent organisations [7] in which agents receive tasks that require services to be executed by an agent itself or by delegating to another agent. Here, reorganisation involves evaluating individual connections between agents based on performance

or potential performance, and creating or removing connections appropriately. The performance of a connection is based on its effect on the load of the agents on either side, the change in the number of messages sent, and cost of reorganising, so that, for example, middlemen in lines of communication are removed. However, the structure of the organisation is never directly analysed, and again the process is based on the system which is application specific.

6 Conclusions

This paper has presented a novel method of reorganisation to optimise system performance, while at the same time preserving organisational structure. While the focus in the paper has been on the case of simple pipelines, the implications of the work, and the more general programme of research in which it is situated, are much more far-reaching. In terms of the specific techniques presented, it is interesting to note that random reorganisation turns out to be most successful at distributing load because it brings about radical rather than incremental changes: this is another case of the simple outperforming the sophisticated in certain conditions. Nevertheless, the broad conclusions that we come to indicate that our techniques are sound in particular aspects as elaborated though the paper, as well as providing a template for further work on other structures. Indeed, as part of a more general programme, this is just the first step. Pipelines are clearly one of the more simple structures that can be analysed, yet this first effort shows the way forward in seeking to develop a general methodology for reorganisation as well as a library of techniques for particular purposes.

References

1. S. Abdallah and V. Lesser. Multiagent reinforcement learning and self-organization in a network of agents. In *In the Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, pages 1–8. ACM Press, 2007.
2. L. Freeman. Centrality in Social Networks: conceptual clarification. *Social Networks*, 1:215–239, 1978/79.
3. M. Gaston and M. Jardins. Agent-Organized Networks for Dynamic Team Formation. In *Autonomous Agents and Multi Agent Systems*, pages 230–237, 2005.
4. C. Gershenson and C. Apostel. Towards Self-organizing Bureaucracies. *International Journal of Public Information Systems*, 2008(1):1–24, 2008.
5. T. Hey. The UK e-Science Core Programme and the Grid. *Future Generation Computer Systems*, 18(8):1017–1031, 2002.
6. T. Hey and A. Trefethe. Cyberinfrastructure for e-Science. *Science*, 308:817–821, 2005.
7. R. Kota, N. Gibbins, and N. Jennings. Decentralised Approaches for Self-Adaptation in Agent Organisations. *ACM Transactions on Autonomous and Adaptive Systems (In Press)*, 2012.
8. S. Lipschutz. *Essential Computer Mathematics*. McGraw-Hill Book Company, 1982.
9. M. Sims, C. Goldman, and V. Lesser. Self-Organization through Bottom-up Coalition Formation. In *Proceedings of The 2nd International Joint Conference On Autonomous Agents and Multi-Agent Systems*, pages 867–874, 2003.