Dynamic Operating Policies for Commercial Hosting Environments

J. Slegers, C. Smith, I. Mitrani, A. van Moorsel and N. Thomas*

July 17, 2006

Abstract

This paper reports on two strands of work that are being undertaken as part of the EPSRC funded DOPCHE project. The paper focuses on open software architectures for dynamic operating policies and a performance model used to find optimal operating policies.

1 Introduction

Amongst the areas of research vital to the development of infrastructure support for eScience is the provision of systems with predictable, differentiated levels of performance and dependability [13]. It is essential that systems be able to provide sustainable levels of service whilst adapting their operating policies to dynamic workloads, resource pools and system configurations. While policy making, prediction and monitoring, and self-tuning infrastructures are available, they exist to different degrees and to different extents, and there is no established framework in which they are (or can be) currently integrated. In future systems it will be imperative that these are provided as fundamental, integrated components of a combined infrastructure.

The success of an eScience infrastructure is based on a number of fundamental requirements, including the ability to provide dynamic, universally available and trusted services. Underpinning such systems is a set of basic functional requirements:

- to understand, capture and define service requirements,
- to verify that the infrastructure is delivering the desired quality of service,
- to dynamically adjust operating policies if the service requirements are not being met.

There is considerable support for the introduction of service-level agreements in Grid computing. These are seen as a mechanism by which work units from a variety of different customers can be arranged and ultimately coordinated through infrastructure-level operating policies. The full exploitation of the infrastructure by a number of different user groups will indeed require several concurrent operating policies running across different virtual organisations and over different geographical sites. It will only be in meeting these predefined policies (and therefore the service-level agreements) that the notion of a trusted ubiquitous system will be established [2, 11].

There are several research areas which support the development and delivery of universally available and trusted services. These include scheduling and reservation, and mechanisms for managing wide-area applications such as caching and data proximity. However, fundamental to these services is the ability to provide a benchmark comparison of one implementation against another, and in so doing be able to recognise and realise the underlying core performance requirements and delivery.

There is a distinct need for research in the areas of:

- systems and application modelling that allow us to predict reliably the behaviour of future eScience infrastructures;
- performance verification that provides evidence-based validation on the delivery of these services;

*School of Computing Science, University of Newcastle, nigel.thomas@ncl.ac.uk
self-tuning and adaptive systems that aim to reduce the cost and complexity of managing the infrastructure in the light of changing user needs and changing infrastructure support.

There is a fundamental need for a coordinated investigation into the relationship between all three work areas in delivering trusted ubiquitous systems, particularly in the light of emerging component technologies.

In this paper we present some initial work from the “Fundamental Computer Science for eScience” funded DOPCHE project. Work at Newcastle is focused in two work packages, one on dynamic software architectures and one on performance-based policy selection.

2 Open Software Architecture for Dynamic Operating Policies

The evolution of the distributed systems paradigm is yielding an increased requirement to facilitate interactions with heterogeneous stateful resources through a uniform interface. Provision of a uniform interface assures interoperability and loose coupling by providing a consistent set of interaction methods used to access and manipulate the state of all constituent resources. The uniform interface therefore seeks to homogenise interaction with heterogeneous resources.

Figure 1 shows an example interaction with the state of a resource through a uniform interface, requesting the value, $V_1$, of some property $P_1$. We denote the request of an interaction as $Rq$, and the response as $Rp$. These requests and responses are specialized by including the type in the form $Rq_{type}$ and $Rp_{type}$ respectively. The interaction methods of the uniform interface enable the state of the Server$_1$ resource to be managed using methods consistent with any other resources implementing this uniform interface.

Enabling interaction with stateful resources through a uniform interface provides opportunities in many deployment environments including self-managing [17] and grid [6] systems. The intrinsically heterogeneous resources in these application domains necessitate consistent interaction methods to allow both the flexible integration of, and intercommunication between participant resources in a self-managing system, and the coordinated resource sharing [6] required in a grid system.

The problem we address here is that of providing an efficient uniform interface to enable interaction with and thus management of stateful resources. The efficiency of a uniform interface shall be judged on the syntax required to convey some semantic notion. Thus, an efficient uniform interface for stateful resource interaction would be semantically complete and require minimal syntax. The term resource is used to describe an identifiable object, and we add to this the notion of a resource being stateful. That is, at some time $t$, the resource $r$ holds some state $S_{t,r}$ characterised by some number of related properties. We define $P_r$ as the set of constituent properties of resource, $r$, and $V$ as the set of all possible values for these properties. Then, omitting the subscript $r$ for readability, we can define the state $S_t$ of a resource formally as:

$$S_t = \{(p, v)|p \in P \land v \in V\}$$

The state $S_t$ represents the evolution of the resource as a result of both time, $T$, and direct interactions, $i \in I$. The evolution implies the transition from some state $S_t$ to state $S_{t+1}$.

Figure 2 shows the transition of a given resource from $S_t$ to $S_{t+1}$ as a result of interaction $i_1$, and from $S_{t+1}$ to $S_{t+2}$ as a result of time. The semantic consequence of the transitions between states is defined by the underlying semantics of the stateful resource. For example, the transition from state $S_t$ to $S_{t+1}$
may amend the value of some property, \( p \in P \) (a change to some entity, \( e \), shall be denoted \( e' \)). The transition from \( S_{t+1} \) to \( S_{t+2} \), conversely, may change the properties composing the resource. The uniform interface should only be concerned with the communication of, and resulting state from the interaction semantics, not their semantic consequence on the represented resource. To represent interactions with a stateful resource, the uniform interface at its most fundamental should supply a set of atomic methods semantically analogous to:

\[
M_{ui} = \{ \text{GET}, \text{SET} \}
\]

These methods enable complete control over the state of some resource, \( r \); facilitating retrieval (\text{GET}) and modification (\text{SET}). Interaction with the resource necessitates the assignment of a unique identifier, \( r_{id} \), such that interactions may be directed at resources using this identifier. More specific functionality can be obtained by providing specialisations of these atomic methods, or conveying increased semantic content to these methods. This yields a trade-off between the cardinality of the interaction method set, \( M_{ui} \), and the syntax required by some \( m \in M_{ui} \). The most efficient solution to the general problem must resolve this trade-off in an optimal manner.

The fundamental issue to be addressed in a solution is the mapping of the uniform interface interaction to the specific resource interaction, which we define formally as:

\[
R_{q_{res}} = UI(R_{q_{ui}}) \quad (1)
\]

\[
R_{p_{ui}} = UI(R_{p_{res}}) \quad (2)
\]

The conciseness with which one can define the mapping, \( UI \), is dictated by the methods chosen to represent \( R_{q_{ui}} \) and \( R_{p_{ui}} \). This mapping must be executed by an engine of some sort, and the greater the semantics the engine is able to derive from some given interaction syntax, the more efficient the engine, and consequently the uniform interface. Increased efficiency of a uniform interface has the subsidiary effects of reducing the burden on both the communication protocol used to transport \( R_{q_{ui}} \) and \( R_{p_{ui}} \) between the source and destination resource, and the footprint of the \( UI \) function at the interaction endpoints.

2.1 Solution in REST Style

The architecture proposed applies the Representational State Transfer (REST) architectural style, put forward by Fielding [7]. REST declares a set of constraints which focus on achieving desirable properties, such as scalability and reliability within a distributed system. Hypertext Transfer Protocol (HTTP) [7], the archetypal application-level protocol for distributed systems, can be used to enforce the constraints of the REST architecture, facilitating stateless, self-descriptive client-server interactions. HTTP, therefore, acts as the basis for the uniform interface of the proposed solution.

The concept of a resource is central to REST, and is formally defined as a ‘temporally varying membership function \( M_r(t) \) which for time \( t \) maps to a set of entities, or values, which are equivalent’ [7]. A resource is conceptual and defined by its mapping to representation at some time \( t \). This definition pertains well to our definition of a stateful resource given above; at time \( t \), \( M_r(t) \) will map the resource \( r \) to
some representation and resource $r$ can be said hold some state $S_t$.

The notion of a conceptual resource allows us to encapsulate the varying representations (states) of a resource into one conceptual entity, drawing many parallels with the WS-Resource construct in WSRF. For the purpose of entity identification, and indeed for the communication of much of the interaction semantics, we use a Uniform Resource Identifier (URI) [18]. At time $t$, $M_r(t) = S_t$, and resolve($URI_1$) $= S_t$, accordingly the current state representation of the resource can be accessed by resolving the identifier, $URI_1$. The state representation is composed of all resource properties, $P$ exposed by a resource at some URI, and these properties may be derived from any number of stateful components behind the uniform interface, which once again is comparable with WS-Resource functionality. We define the methods of a uniform interface using HTTP as follows:

$$M_{http} = \{GET, POST, PUT, DELETE\}$$

The interaction methods offered by this uniform interface are defined by HTTP. These methods enable access to (GET), and manipulation of (POST, PUT, and DELETE) any resource residing at some URI. These methods are communicated using standard HTTP requests, and thus communication is syntactically uncomplicated. The manipulation methods could conceivably be combined into a single POST method, as creation (PUT) and deletion (DELETE) of state could be seen as forms of amendment. This would concatenate the method set, but would require additional semantics to be conveyed implicitly in the request body, rather than explicitly in the method line of the request. For syntactic and semantic simplicity we retain all standard HTTP methods shown above.

Introspection on, and amendment to the resource state can be performed through the utilisation of basic HTTP methods and resource URI. The resulting state of a transition can be defined formally as: $S_{t+1} = (S_t \cup \{(p_i, v_i)\}) \setminus \{(p_i, v_i)\}$, where $p_i \in P_r \land p_i \in P_r$. State transition is where this uniform interface greatly simplifies any previous solutions.

Any of the interaction methods, $m \in M_{http}$, may include metadata in a response, which is used for the communication of additional information regarding the state. For example, included in the metadata of a GET would be the URIs for access to, and manipulation of component properties of the state, using the standard interaction methods. Supplementary metadata may include XML Schema [19] to define the structure of interaction data. Properties are therefore treated as conceptual resources, and we can partition state into components using URIs, simplifying interactions markedly. For instance (Figure 5), if some resource has properties $\{p_1, p_2\}$, GET to $URI_1$ will return $S_t = \{(p_1, v_1), (p_2, v_2)\} \cup \{URI_2, URI_3\}$. The URIs are not parts of the state of resource $r$, they are simply metadata to communicate how component properties may be changed. Interactions can then be directed at these URIs to access or manipulate the associated property.
roduce a new managed resource instance and be returned the URI of the newly created resource for future interactions. The state representation of the management resource would be amended accordingly. This resource notion enables the modelling of any identifiable object whether conceptual or concrete, giving us unbounded modelling flexibility.

\[ R_{res} = HTTP(R_{http}) \]  

\[ R_{http} = HTTP(R_{pres}) \]  

The mapping function, HTTP simply inspects the destination URI and HTTP method of the incoming request. If such a resource exists, and this method of interaction is supported, the HTTP request is forwarded to the resource to execute the internal semantics and respond. If URI does not exist or the method of interaction is not supported, the conventional HTTP response codes are used. We have thus improved the conciseness of the mapping function markedly. All necessary information for interaction is packaged into the HTTP request and URI, no additional contextual information is held on the server relating to a given sequence of interactions. Any context concerning a given (sequence of) interaction(s) is incorporated into the URI and held on the client-side. This adheres to the REST constraint of stateless interactions. The abstraction provided by the concept of a resource in REST enables functionality analogous to the WS-Resource in WSRF. The low level composition of state is again shifted outside of the scope of the uniform interface, relieving much of the semantic and syntactic burden and enabling increasingly flexible notions to be represented as resources. By exploiting the semantics of HTTP and URI, the beneficial effects of abstraction on the uniform interface have been further developed. There have been other attempts to utilise HTTP for the expression of resource interaction semantics [16, 3], but none have fully exploited the benefits of HTTP and URI semantics. Concentration has been on manipulation of HTTP requests and utilisation of URIs to communicate the semantics of some underlying management standard, for instance SNMP [16]. Such solutions are ineffective as they must undergo two stages of processing, from HTTP and URI to the underlying management standard and from this standard to the resource-specific interaction. The resulting architectures therefore demonstrate many of the restrictions of the underlying management standards, albeit while utilising a different communication medium. The REST architecture we propose, through exploitation of HTTP method and URI semantics, enables semantically complete interaction with stateful resources. The abstraction provided by the resource concept enables the reduction in cardinality of the interaction method set, and the utilisation of HTTP methods to convey interaction semantics enables syntactic brevity in interaction. Instead of placing a large amount of syntactic burden on the transport protocol, and mapping function to convey the semantics of the interaction, we are deriving all required semantics from the URI, HTTP method, and any content associated with the request (in some standard data format).

3 Optimal resource provisioning of application hosting in a changing environment

Recent developments in distributed and grid computing have facilitated the hosting of service provisioning systems on clusters of computers. Users do not have to specify the server on which their requests (or 'jobs') are going to be executed. Rather, jobs of different types are submitted to a central dispatcher, which sends them for execution to one of the available servers. Typically, the job streams are bursty, i.e. they consist of alternating 'on' and 'off' periods during which demands of the corresponding type do and do not arrive.

In such an environment it is important, both to the users and the service provider, to have an efficient policy for allocating servers to the various job types. One may consider a static policy whereby a fixed number of servers is assigned to each job type, regardless of queue sizes or phases of arrival streams. Alternatively, the policy may be dynamic and allow servers to be reallocated from one type of service to
another when the former becomes under-subscribed and the latter over-subscribed. However, each server reconfiguration takes time, and during it the server is not available to run jobs; hence, a dynamic policy must involve a careful calculation of possible gains and losses.

The purpose of this work is to (i) provide a computational procedure for determining the optimal static allocation policy and (ii) suggest acceptable heuristic policies for dynamic server reconfiguration. In order to achieve (i), an exact solution is obtained for an isolated queue with \( n \) parallel servers and an on/off source. The dynamic heuristics are evaluated by simulation.

The problem described here has not, to our knowledge, been addressed before. Much of the server allocation literature deals with polling systems, where a single server attends to several queues \([4, 5, 8, 9, 10]\). Even in those cases it has been found that the presence of non-zero switching times makes the optimal policy very difficult to characterise and necessitates the consideration of heuristics. The only general result for multiple servers concerns the case of Poisson arrivals and no switching times or costs: then the \( c_i \)-rule is optimal, i.e. the best policy is to give absolute preemptive priority to the job type for which the product of holding cost and service rate is largest (Buyukkoc et al \([1]\)).

A model similar to ours was examined by Palmer and Mitrani \([12]\); however, there all arrival processes were assumed to be Poisson; also, the static allocation was not done in an optimal manner. The novelty of the present study lies in the inclusion of on/off sources, the computation of the optimal static policy and the introduction of new dynamic heuristics.

### 3.1 The model

The system contains \( N \) servers, each of which may be allocated to the service of any of \( M \) job types. There is a separate unbounded queue for each type. Jobs of type \( i \) arrive according to an independent interrupted Poisson process with on-periods distributed exponentially with mean \( 1/\xi_i \), off-periods distributed exponentially with mean \( 1/\eta_i \) and arrival rate during on-periods \( \lambda_i \) \((i = 1, 2, ..., M)\). The required service times for type \( i \) are distributed exponentially with mean \( 1/\mu_i \). This model is illustrated in Figure 6.

![Figure 6: Heterogeneous clusters with on/off sources](image_url)

Any of queue \( i \)'s servers may at any time be switched to queue \( j \): the reconfiguration period, during which the server cannot serve jobs, is distributed exponentially with mean \( 1/\zeta_{i,j} \). If a service is preempted by the switch, it is eventually resumed from the point of interruption.

The cost of keeping a type \( i \) job in the system is \( c_i \) per unit time \((i = 1, 2, ..., M)\). These 'holding' costs reflect the relative importance, or willingness to wait, of the \( M \) job types. The system performance is measured by the total average cost, \( C \), incurred per unit time:

\[
C = \sum_{i=1}^{N} c_i L_i ,
\]

where \( L_i \) is the steady-state average number of type \( i \) jobs present. Those quantities depend, of course, on the server allocation policy.

In principle, it is possible to compute the optimal dynamic switching policy by treating the model as a Markov decision process and solving the corresponding dynamic programming equations. However, such a computation is tractable only for very small systems. What makes the problem difficult is the size of the state space one has to deal with. The system state at any point in time is described by a quadruple, \( S = (j, n, u, m) \), where \( j \) is a vector whose \( i \)th
element, \( j_i \), is the number of jobs in queue \( i \) (including the jobs in service); \( n \) is a vector whose \( i \)th element, \( n_i \), is the number of servers currently assigned to queue \( i \); \( u \) is a vector whose \( i \)th element, \( u_i \), is 0 if the \( i \)th arrival process is in an off-period, 1 if it is on; \( m \) is a matrix whose element \( m_{i,k} \) is the number of servers currently being switched from queue \( i \) to queue \( k \). The possible actions that the policy may take in each state are to do nothing or to initiate a switch of a server from queue \( i \) to queue \( k \).

A numerical procedure to determine the optimal policy would involve truncating the queue sizes to some reasonable level, discretizing the time parameter through uniformization and then applying either policy improvement or value iterations (e.g., see [14, 15]). It is readily appreciated that the computational complexity of that task grows very quickly with the number of queues, \( M \), the number of servers, \( N \), and the truncation level. For that reason, we have concentrated on determining the optimal static allocation policy (which does not involve switching) and comparing its performance with that of some dynamic heuristics.

### 3.2 Initial results

Figure 7 shows the total (i.e. for both job types combined) average (i.e. over time) cost, over a period of \( T = 10000 \), of a system with 2 job types and 20 servers. Both the job types are on for half the time but one of them has a cycle lasting (in mean) 50 and the other 100. The holding cost for the first job type is 1, for the second it is 2. In all other aspects they are identical and the rate at which jobs are completed per server is 1. The x-axis represents increased load, with the rate of job arrival increasing from 15 to 19.5. This means the system is only stable on average. The switching time and cost is set to be very small in this case. Heuristic 1 is making switching decisions based on a fluid cost approximation until the next time the queue empties. The assumption here being that job types that are on, remain on and vice versa. Heuristic 2 is the same as above, but using the average load to phase the on-off periods out of the model.

Because of time constraints, these are averages over a small number of simulations. This explains some big variations, especially in the static cases. Note that it is probably not a coincidence that the dynamic simulations seem to fluctuate much less, their very nature makes them more capable of dealing with more extreme events. Note that the dynamic policies are a big improvement over the static ones studied previously [12].

### 4 Conclusions

In this paper we have presented some preliminary results concerning issues relating to dynamic deployment within commercial hosting environments. These results show the suitability of using the REST architectural style for developing an open software architecture for dynamic operating policies. In addition we have shown some initial results from modelling experiments using stochastic modelling techniques to derive improved policy decisions. These results show that where the requests for a service fluctuate, taking this additional information into account when assigning servers can have a significant benefit.

A substantial amount of further work remains to be done within the DOPCHE project in developing optimal and heuristic policies and in developing the
architectural support to implement these policies in practise.

References


