Service and Resource Discovery in Cycle-Sharing Environments with a Utility Algebra

João Nuno Silva  Paulo Ferreira  Luís Veiga
INESC-ID / Instituto Superior Técnico - Technical University of Lisbon
Rua Alves Redol Nº 9, 1000 Lisboa, Portugal
Email: {joao.n.silva, paulo.ferreira, luis.veiga}@inesc-id.pt

Abstract—The Internet has witnessed a steady and widespread increase in available idle computing cycles and computing resources in general. Such available cycles simultaneously allow and foster the increase in development of existing and new computationally demanding applications, driven by algorithm complexity, intensive data processing, or both. Available cycles may be harvested from several scenarios, ranging from college or office LANs, cluster, grid and utility or cloud computing infrastructures, to peer-to-peer overlay networks.

Existing resource discovery protocols have a number of shortcomings for the existing variety of cycle sharing scenarios. They either i) were designed to return only a binary answer stating whether a remote computer fulfills the requirements, ii) rely on centralized schedulers (or coherently replicated) that are impractical in certain environments such as peer-to-peer computing, iii) they are not extensible as it is impossible to define new resources to be discovered and evaluated or new ways to evaluate them.

In this paper we present a novel, extensible, expressive, and flexible requirement specification algebra and resource discovery middleware. Besides standard resources (CPU, memory, network bandwidth, . . .), application developers may define new resource requirements and new ways to evaluate them. Application programmers can write complex requirements (that evaluate several resources) using fuzzy logic operators. Each resource evaluation (either standard or specially coded) returns a value between 0.0 and 1.0 stating the capacity to (partially) fulfill the requirement, considering client-specific utility depreciation (i.e., partial-utility, a downgraded measure of how the user assesses the available resources) and policies for combined utility evaluation. By comparing the values obtained from the various hosts, it is possible to precisely know which ones best fulfill each client’s needs, regarding a set of required resources.

I. INTRODUCTION

Currently, there is increasingly greater offer of computing cycles and computing resources in general, whether paid or free. There is a myriad of alternative approaches and technologies encompassing: computing clusters (possibly virtualized), idle time of computers in office or college LANs, academic and scientific institutional grids creating virtual organizations with coordinated scheduling, utility and cloud computing infrastructures, distributed computing on opportunistic or desktop grids, and peer-to-peer (fully decentralized or federated) cycle-sharing topologies. At the same time, and also motivated by that, there are more and more applications designed to take advantage of such available resources.

Regardless of the underlying cycle-providing infrastructure and the applications, resource discovery (e.g., CPU, memory, bandwidth, specific hardware installed) is a key enabling mechanism. It serves a double purpose of enabling application execution as well as driving overall system usefulness. Service discovery, viewed broadly, may be considered as a higher-level abstraction of resource discovery where resources also include: available services and installed applications at the host. Discovery should also be driven by other criteria regarding software such as versions, packages, libraries, licences, etc.

Today, there are many systems supporting the discovery of resources; e.g., grid infrastructure schedulers [1], [2], [3], resource monitoring systems [4] with possible resource aggregation [5], [6], [7], service discovery protocols [8], [9], and web service discovery [10]. These protocols and systems have a number of shortcomings for the wide variety of present cycle sharing scenarios. They either i) were designed to return only a binary answer stating whether a remote computer fulfills the requirements, ii) rely on centralized schedulers (or coherently replicated) that are impractical in certain environments such as peer-to-peer computing, iii) they are not extensible as it is impossible to define new types of resources to be discovered and evaluated, or iv) they are inflexible in the sense that users and administrators are unable to set alternative ways to evaluate and assess available resources.

Therefore, we designed a middleware platform (STARC) to make resource discovery more adaptive via extensibility (ability to incorporate new resources) and increased flexibility (expressiveness in requirement description, utility function depreciation on partial fulfillment, and employing fuzzy-logic to combine multiple requirements). STARC is able to interface with different network topologies e.g., LAN, peer-to-peer (P2P). STARC extensibility stems not only from allowing the evaluation of most usual resources (e.g., CPU speed and number of cores, memory) but also from the dynamic inclusion of new characteristics (presence of specific applications, services, libraries, licenses or hard-
ware) to be evaluated.

Regarding flexibility, STARC uses XML files to describe application requirements stating, with logic operators, the relation between the several resource characteristics that are relevant, with associated value ranges, and utility depreciations in the case of only partial fulfillment being possible (i.e., sets of resource availability ranges and associated utility depreciation or partial-utility). Hosts may return information ranging from no availability (0.0), to requirements fully met (1.0). If requirements are only partially met, a value between 0.0 and 1.0 is returned (partial-utility), taking into account the partial fulfillment ranges and associated utility depreciations (not necessarily strict linear mappings), as well as evaluation policies provided by the client. Utility depreciations are also applicable to individual resource alternatives not enclosed within ranges (e.g., OS family).

In the next section we present some available systems that allow the evaluation of remote computer resources, comparing their shortcoming with our solution. In Section III we present STARC architecture, components, and resource discovery protocol employing a utility algebra that aggregates the utility of each individual resource included in a resource discovery request. In Sections IV and V we describe STARC implementation and evaluation. Section VI closes the paper with conclusions and future work.

II. RELATED WORK

Currently, there are a number of systems that allow the discovery of resources in network-connected computers. Such systems fall into the following categories: cycle sharing systems, resource management in grid infra-structures, service discovery protocols, or utility-based scheduling. In this section we describe these systems and present their limitations regarding the intended flexibility, expressiveness and extensibility.

A. Cycle sharing systems


In either set of systems the processing power or other relevant resources are not taken into consideration. In projects such as BOINC or CCOF, only the processor state is relevant when selecting the remote host that will execute the code. This solution is easy to implement, fair to the owner of the remote computer, but may slow the overall application, while being restrictive by not considering other resources and partial fulfillment of the requested resources.

B. Grid Resource Management

In order to optimize the use of grid resources it is necessary to choose the hosts that best answer to the resource requirements of applications. The work described in [14] offers a study on the performance and scalability of most widely deployed approaches to resource monitoring in existing grid middleware (MDS, R-GMA and Hawkeye). They are found to offer similar functionality and performance with good scaling behavior w.r.t. the number of clients making simultaneous requests to the system. Next, we describe some important features of resource discovery in Grid infrastructures and their limitations w.r.t. our proposal. MDS4 [5] describes a Resource Aggregator Framework that could allow implementations to calculate combined utility of a set of resources. To the best of our knowledge, we have found none that implements the utility algebra we propose in this paper and described in Section III.

Both Condor [15], Condor-G [3] and Legion [16] provide mechanisms to state what requirements must be met in order to execute efficiently some code. Legion uses Collections [1]: repositories of information describing the resources of the system, that are queried to find the host having the required resources. Collection queries are built with the usual relational and logical operators. Condor, Condor-G and Hawkeye use ClassAd [2], [3], [7] objects in order to describe computing nodes’ resources (providers) and define requirements and characteristics of programs (customers). ClassAd is a dictionary where each attribute has a corresponding expression. These attributes describe the owner of the ClassAd. Matchmaking of requirements and resources is accomplished using customers’ and providers’ ClassAds. The rank attribute allows ordering customers and providers with matching attributes. Such systems allow the description of application requirements and the discovery of a computer that supports its efficient execution. However, if a host does not completely satisfies a requirement, it is simply considered as not eligible. Furthermore, aggregation of several ClassAd with configurable weights by users is not available.

Current Globus GRAM[17] implementations, as described in [18], suffer from the same problems: usually, a fixed infrastructure is needed to collect and monitor resource information. Although this information may be replicated for higher reliability and throughput, this is often done resorting to additional institutional servers, frequently co-located. The set of resources to monitor, evaluate, and aggregate, although configurable, is pre-determined within a virtual organization. This renders Grid Resource Management systems inadequate to dynamic environments, where there is no centralized infrastructure and the resources to evaluate are highly variable.

In the work described in [19], clients are able to select hosts based on user-provided ranking expressions, in order to evaluate resources considering peak and off-peak
periods, sustained availability over a period of time, time of day. In [20], an adaptive resource broker estimates job completion time based on periodically sampling available resources (CPU) in order to trigger job migration. However, both systems are designed to speedup execution and trigger job migration, bearing no information on how multiple resource requirements (besides CPU) and their partial fulfillment could be described and evaluated. Furthermore, in the scenario targeted by STARC (embarrassingly parallel applications), tasks can be made much smaller and job migration becomes almost a non-requirement.

C. Service Discovery Protocols

Service discovery protocols, such as SSDP [8] and SLP [9], allow a client computer to search and discover other computers that are service providers. These protocols allow a service provider to advertise its presence. In this advertisement, the service provider sends a description of the exported service. A client receiving such a message compares the service description with the desired service requirement. If the requirements match the service description, the client can start using the service. The discovery of the services can also be initiated by the client, by broadcasting a message with the requirements the service provider must comply to. Every service provider that has a service that matches the requirements answers with its identification.

SLP [9] also allows the existence of Directory Agents, central servers that are contacted by the client when looking for services and by the Service Providers to publicize its services. Web service discovery [10] can be employed in distributed computing scenarios to find locations where remote functionality may be executed. Even if it is possible to find devices and hosts that have a certain service, it is not possible to easily evaluate the available resources given that these protocols were designed to ease the discovery (i.e., mere presence) of services. The requirements are tested against the static characteristics of the service, not allowing the evaluation of available resources at the computer. Whenever several versions of the same service could be used by the client, these protocols still do not allow the association of utility depreciations to outdated versions (i.e., accepting some outdated versions while preferring current one), therefore being inflexible.

D. Utility-based Scheduling

There are systems that perform scheduling based on utility functions [21], [22], [23], [24] aiming at maximizing overall system utility or usefulness. However, these solutions suffer from a number of drawbacks. They assume there is a coordinator or scheduler node that receives and processes all resource discovery requests and therefore is able to employ complex pricing models that globally optimize resource usage as well as maximize user utility. The way they incorporate utility functions is rather inflexible as they either:

i) consider only complete requirement fulfillment and award it a utility (possibly weighted), or ii) they assume a nearly constant elasticity model where the lack of availability of one resource can be supplantcd by a surplus of others for equal utility (clearly a system-centric approach and not user-driven), or iii) provide no support for hints regarding resource or service adaptation in case of partially unfulfilled requirements.

III. SERVICE AND RESOURCE DISCOVERY

Applications need to discover, evaluate and select the resources present and available in remote computers as well as services and software provided by them. This is achieved via a middleware platform (STARC), an assemblage of components (STARC daemons), that execute both in clients and in resource providers, all regarded as peers. Each daemon handles all requirement evaluation requests: those generated from a local application and remote requests generated by other remote daemons. The architecture of STARC is presented in Fig. 1 and is described briefly next.

To use STARC, an application must provide to the local STARC daemon a XML Requirement File containing a logical description of the hardware and software requirements and alternatives (Step 1 in Fig. 1). The STARC daemon reads the requirements and executes the relevant Environment Probing Classes (Step 2) in order to know how the resources fulfill the requirements. The logical descriptions are evaluated against the values returned by the Environment Probing Classes according to specified partial-utility functions and combined utility evaluation policies defined by a utility algebra (more details in Section III-B2).

After local resource evaluation, if the request was originated from a local application, the STARC daemon contacts the remote daemons (Steps 3) by means of the Communication component. Each contacted daemon evaluates the requirements (Step 4) and returns the resulting value (Step 5). The Remote Host Discovery module finds hosts that have a STARC daemon running. This component abstracts the middleware from different network topologies. It interfaces with the rest of the middleware uniquely by providing for each request a list of available hosts. These hosts are later contacted by the Communication component. Further details on remote host discovery when addressing different network topologies (e.g., LAN, coordinator/scheduler-based grids or virtual organizations, peer-to-peer overlays) are addressed in Section IV.

A. Requirement Specification

In order to use STARC, the user or programmer writes a XML file stating application requirements and feeds it to a locally running STARC daemon by invocation of a single method. The XML file defining application requirements has the syntax presented in Fig. 2. These requirements can be physical resources (e.g., available memory, processor speed,
GPU installed), installed software (e.g., certain operating systems, virtual machines, libraries, helping applications) or availability of specific services (e.g., logging, checkpointing, specific web services). On completion of discovery, the application will receive a list identifying available hosts and how fit they are to fulfill those requirements.

An example requirement is presented in Fig. 3 and will be used as a prototypical example hereafter. A requirement can refer a simple resource (CPU in Fig. 3) or it can be a complex composition of other requirements using logical operators. The composition of requirements is accomplished by using usual logical operators (and, or, not) whose evaluation will be described later in the Section.

In order to precisely define a required resource, it is necessary to state its name and, if necessary, the ranges of resource values and associated utility depreciation, in the config element. This config element (e.g., processorCores, processorSpeed, version in Fig. 3) will be used during resource evaluation. From this expression it is possible to know whether the computer is adequate.

Using the prescribed syntax it is possible to write complex requirements with the conjunction or disjunction of the characteristics of the resources. For instance, it is possible to state that a program needs a certain number of processor cores, with specified speed, and either one of two libraries, as illustrated in Fig. 3. In this example, we want to know if a certain computer has an ideal configuration of at least 16 available processor cores preferably with speed of at least 3000 MHz, and has either the NumPy (an updated version) or the PyGPU library installed. In this case, two versions of the application can be uploaded to hosts, one using the NumPy library and the other using PyGPU.

To state this information we use the or and the and logical operators and three different resources. Inside the CPU resource element we state the required configuration for processorCores and processorSpeed (an implicit conjunction): we use the range, minnumber XML elements to specify that information.

**B. Utility Algebra**

In addition to specifying requirements, XML files allow users to employ a utility algebra, described next, comprised of: i) ranges of resource values with associated client-specific utility depreciations (partial-utility), ii) alternative resource options with associated utility depreciation, and iii) policies for combined satisfaction evaluation.

1) User-defined Partial-Utility Criteria: If a host can completely fulfill the requirement, it returns the highest possible value (i.e., 1.0 which is made explicit in the XML files but is assumed if a range has no utility defined), it surely is among those that best fit the requirement. However, this kind of binary answer to the evaluation of requirements is rather inflexible, therefore a host that cannot fulfill a requirement completely is not necessarily considered as having utility 0.0. As it returns information (e.g., a numerical value) stating its partial capacity to meet the requirement, this can be evaluated against a client-specific utility depreciation. Such depreciation is supplied in the requirement XML file provided by the client, stating how (un-)willing the client is to accept a lower amount or quality of the required resources.

Thus, regarding the prototypical example, the same given host CPU configuration (processor cores and speed) and the same CPU evaluation code will produce different levels of
However, the overall system is rendered more flexible and assurance of utility depreciation, that can be used to drive resource specialization. Moreover, XML requirement files issued by clients contain partial fulfillment w.r.t. resource availability (partial-utility) of individual options (e.g., operating system, installed operating system (e.g., according to the specifics of each client’s request). These include render application version, minimum number of cores required for the job to complete in useful time span, and number of cores above which the user has not a keen interest in. As mentioned earlier, expressing partial-utility using ranges allows non-linear depreciation. 

2) Policies for Combined Satisfaction Evaluation: Naturally, given a XML requirement file with a set of resource and service requirements, it is necessary to select the host(s) best fit for it. This must take into account not only the (partial-)utility w.r.t. each requirement but also an evaluation of combined satisfaction regarding the set of requirements as a whole, i.e. a global (possibly weighted) measure of utility resulting from aggregating each set of (possibly adapted) resources proposed by hosts.

As discussed earlier in Section II, resource management and job scheduling in grids do not consider partial requirement satisfaction and employ complex utility functions to optimize scheduling within a virtual organization, i.e., considering system utility based on budgets and deadlines. With STARC, we propose a utility algebra that takes user-perceived values of partial-utility into account in a flexible and expressive manner without resorting to complex tailored economic models that often also assume only elasticity of substitution. The evaluation of these logical expressions is accomplished resorting to the following operators: and, or and not. The or operator always returns the largest operand, the resource alternative providing the highest utility. With the and operator, the user wants all requirements to be met, so the result of this operator is an aggregate value not allowing a user to specifically value a resource negatively.

```
<requirement policy="strict"><and>
  <requirement><resource name="CPU">
    <config><processorCores>
      <range><minnumber>10</minnumber><range>
        <util>0.5</util>
      </range>
    </config>
  </resource>
  <requirement><resource name="NumPy">
    <config><version>
      <range><minnumber>2400</minnumber><util>0</util>
    </range>
  </resource>
</and>
```

Figure 3. Prototypical example of XML requirements description, including: i) nonlinear depreciation of utility values in processorCores and version number for partial fulfillment ranges, and ii) the combined utility evaluation policy selected (strict) described in detail in Section III-B.
This may happen when a user wants that a specific undesired resource alternative may actually contribute negatively to the combined utility, in a multiple requirement request.

In the same manner as the Environment Probing Classes return values between 0.0 and 1.0, these logical operators return values in the same range, indicating how capable a host is to satisfy the requirements. Using these operators and comparing the values returned from the evaluation of a requirement on different computers, it is possible to find the one(s) more capable: the one(s) with the highest requirement evaluation value(s).

STARCS allows users to select how combined requirements (operator and), alternatives (operator or), and disapproval (operator not) are evaluated according to a number of different policies described next (priority, strict, balanced, elastic, and userclass as a default value mapped to one of the others as decided by the system). Each policy is inspired by a specific scenario and targets a class of intended users (summarized in Table I). A policy embodies a specific aim or goal on how to engage available resources to fulfill resource discovery requirements. This aim is implemented in the way aggregate utilities are calculated in order to evaluate how a set of available resources satisfies a given request (i.e., a numeric measure of combined satisfaction).

Nonetheless, a daemon running at a host may override the selected policies offering different quality-of-service considering user information (e.g., user class, rank, reputation) while notifying the requesting user. Each of the policies is described in the reminder of the Section, by increasing order of flexibility, in essence, user willingness to accept lack of one required resource in exchange for increased availability of others.

- **Priority**: This policy enforces guarantees of resource selection according to user-supplied priorities. It aims at satisfying lexicographic preferences that are not representable by utility functions based on formulæ. This happens when a user wants to prioritize alternatives to resource requirements and is not willing to accept hosts where one or more of the required resources are absent (employing operator and). This is the most rigid of evaluation policies and should only be used in service-critical scenarios. Requirements

\[
\begin{align*}
\text{and}(A, B) &= \min\{A, B\} \\
\text{or}(A, B) &= \max\{A, B\} \\
\text{not}(A) &= (1 - A)
\end{align*}
\]

Figure 4. Extended prototypical example of XML requirements description, including: i) nonlinear depreciation of utility values in processorCores and processorSpeed number for partial fulfillment ranges, ii) utility depreciation associated with individual alternative options (OSFamily), iii) combination of both (renderer application and utilities), and iv) combined utility evaluation policy selected (elastic)

Figure 5. Zadeh Logical operators
Table I

<table>
<thead>
<tr>
<th>Policy</th>
<th>Intended users</th>
<th>Calculation</th>
<th>AND (aggregation)</th>
<th>OR (adaptation)</th>
<th>NOT (disapproval)</th>
</tr>
</thead>
<tbody>
<tr>
<td>priority</td>
<td>administrators</td>
<td>lexicographic</td>
<td>must all occur</td>
<td>follow priority list</td>
<td>reject and fail</td>
</tr>
<tr>
<td>strict</td>
<td>SLA users</td>
<td>Zadeh fuzzy logic</td>
<td>min max</td>
<td>complement</td>
<td>(1-x)</td>
</tr>
<tr>
<td>balanced</td>
<td>registered members</td>
<td>geometric average</td>
<td>max product</td>
<td>inverse</td>
<td>(1/x)</td>
</tr>
<tr>
<td>elastic</td>
<td>best-effort</td>
<td>arithmetic average</td>
<td>sum (averaged)</td>
<td>max opposite</td>
<td>(-x)</td>
</tr>
</tbody>
</table>

combined with operator or (in this case representing a simple alternative) will be tried in sequence by order of appearance (priority) in the XML file. Operator not results in the rejection of any host where the resource resides (normally, to reject a certain architecture or software considered unfit or unsafe).

- **Strict**: This policy aims at minimizing dissatisfaction on every resource requirement. Thus, at each host, for a given set of requirements, when complete fulfillment is not possible, each requirement will be assessed with its partial utility. Then, the combined evaluation will result in the lowest of the combined operands. This follows the fuzzy logic Zadeh [25] depicted in Fig. 5. Thus, operator and returns the minimum partial utility in a set of requirements and operator or returns the maximum. Operator not returns the complementary utility (1-x), typically as a means to lower combined utility (since and always returns the minimum value). A common scenario for this policy would be the case of users with service-level agreements that, even when a single one is partially unmet, impose penalties in the aggregate utility (once again due to and returning the product). This way, if a resource requirement is only satisfied in half, this will bring the overall combined utility also to half. With this policy, hosts that can fully satisfy certain requirements but can only produce a very poor alternative in others, will not be so highly considered. Hosts that partially satisfy all requirements at a good level will be preferred instead. Of course, hosts with well balanced yet low availability in most/all resources should still be awarded lower utility. Typical users will be regular members of a virtual organization such as in a Grid.

- **Balanced**: This policy aims at selecting the most balanced host, that is, the host whose combination of available resources satisfies user requests in a more favorable and balanced manner. This policy is based on geometric averages. Operator and will return the product of all partial utilities and operator not its inverse. A common scenario for this policy would be the case of registered users that, when a resource is partially unmet, this will impose a proportional penalty in the aggregate utility (once again due to and returning the product). This way, if a resource requirement is only satisfied in half, this will bring the overall combined utility also to half. With this policy, hosts that can fully satisfy certain requirements but can only produce a very poor alternative in others, will not be so highly considered. Hosts that partially satisfy all requirements at a good level will be preferred instead. Of course, hosts with well balanced yet low availability in most/all resources should still be awarded lower utility. Typical users will be regular members of a virtual organization such as in a Grid.

- **Elastic**: This policy aims at maximizing engagement of resources by the system, employing an eager-like approach by assuming full elasticity or resources, that is, any non-satisfied resource can be compensated by a good utility in another resource. This is usually not the case since resources are not interchangeable (e.g., CPU for memory and vice-versa). Thus, this policy simply offers a best-effort approach to requirement fulfilling. Combined evaluation is based then on simple arithmetic averages. Operator and will return the averaged sum of all partial utilities and operator not its opposite (additive inverse). With this approach, the system can take advantage of any resource because unfulfillment of a requirement can never bring down the utility value resulting from a combined requirement evaluation. This policy may be used for non-registered users in a peer-to-peer cycle sharing scenario. This is the scenario for the requirement specification depicted in Fig. 4.

**IV. IMPLEMENTATION**

STARC is implemented in Python. The Python standard XML processing library is used to parse the requirement files and generate an internal representation of the logical expressions. The Communication module uses the Pyro [26] remote object invocation library. To simplify our system, the
interaction between any application and its local STARC daemon is also made by means of a Pyro invocation. Although we used Python, any other language that supports dynamic code loading, remote method invocation, and remote code loading could have been used.

In order to easily extend our system we use the reflective mechanisms and class loading provided by the Python runtime. The reflective dynamic loading of the Environment Probing Classes supports the addition of types of new resources to be evaluated. Any programmer can define new proprietary, user-specific, or compound resources to be evaluated, and software or services to be discovered simply by developing a new Environment Probing Class.

All Environment Probing Classes implement a predefined interface composed of a constructor with no arguments and a method called evalResource. This method is invoked by the Requirement Evaluator and receives as a parameter the config XML snippet described in Section III-A. The name of Environment Probing Classes is obtained from the name element present in the XML (Fig. 2). This name is used to dynamically load the corresponding module from disk. When necessary, this class is instantiated and its code executed. If not present in the remote host, the Resource Probing Classes can be transferred from the local computer that initiated the requirement evaluation and executed in a restricted safe environment, allowing the evaluation of specific user requirements. If for some reason the Environment Probing Class associated to a resource can not be executed, the evaluation of that resource requirement will return 0.0, meaning the computer does not have the resources to meet that requirement.

We have developed a set of standard Environment Probing Classes present in every host running STARC allowing the evaluation of a number of most used resources: CPU family, cores and frequency, cache size, memory size, available memory, network link speed available. In the implementation of these classes we used the WBEM [27] capabilities provided by Windows and the /proc file system [28] in Linux.

Although it is out of the scope of the paper, existing ontology specifications (e.g., RDF [29], OWL [30]) can be employed in order to enforce the semantic consistency of actual classifications and meaning of resource and service descriptions in large scale deployment scenarios, as well as describing groups of akin resources and services.

A. Remote Host Discovery

Regarding discovery of remote hosts for resource evaluation, we identify three main network topologies of increasing scale and membership flexibility/variation: i) clusters and LAN; ii) grid-based virtual organizations, and iii) peer-to-peer cycle-sharing desktop grids.

- **Cluster/LAN Scenarios:** In the LAN implementation, the Remote Host Discovery module finds remote computers in the same sub-network. Any other computer discovery protocols could have been used (e.g., Jini, UPnP). Each host evaluates the requirements against its resources and returns the resulting partial-utility value. These are combined with host identification in a bounded ordered list by the local STARC daemon. Most applications need only pick the first in list or iterate over it for parallel scheduling.

- **Grid Infrastructures:** We are currently implementing a STARC module within the framework of MDS4 [5] with a set of Resource Providers (namely residing to Hawkeye [7] for monitoring node resources and availability) and integrating the utility algebra in MDS4 Aggregator Framework (including partial-utility evaluation and policy enforcement). As mentioned in the related work section, this approach and architecture have been previously evaluated and determined scalable [14]. Therefore, we only need to ensure that the algebra evaluation does not impose excessive overhead (more details in the next Section). A similar approach could be pursued for integration with R-GMA [6] or other meta-schedulers, such as Condor-G [3].

- **P2P Cycle Sharing:** Regarding cycle sharing peer-to-peer infrastructures [31], the set of hosts made available to STARC to evaluate should be reduced and not the entire P2P population as this would preclude scalability. Therefore, we have integrated STARC as an additional service on a peer-to-peer cycle sharing system [32], in order to evaluate only two complementary sets of hosts: direct neighbors in routing tables, and those returned by a lower-level resource discovery mechanism seeking CPU, memory and bandwidth.

B. Security

STARC middleware has a small code footprint (below 500 KB) and is executed at host nodes within the boundaries of a virtual machine. Therefore, its access to local resources (e.g., file system, communication, etc.) can be limited and configured. This extends to the vast majority of Environment Probing Classes. If one needs to access the native system directly, it can be subject to individual configuration and will not be executed without user’s authorization. Finally, since STARC only schedules tasks that execute applications already installed at host nodes, it does not entail additional security threats than those already inherited from the actual applications the user has already decided to install.

The dynamic loading of probing classes has two different usage scenarios. One where the probing classes are stored
Table II

<table>
<thead>
<tr>
<th>Approach</th>
<th>Local Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>WBEM</td>
<td>36.3</td>
</tr>
<tr>
<td>Algebra</td>
<td>38.2</td>
</tr>
<tr>
<td>STARC (loopback)</td>
<td>40.1</td>
</tr>
<tr>
<td>SNMP (loopback)</td>
<td>110.01</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Approach</th>
<th>loopback</th>
<th>no upload</th>
<th>upload</th>
</tr>
</thead>
<tbody>
<tr>
<td>STARC (simple)</td>
<td>40.1</td>
<td>41.0</td>
<td>110.0</td>
</tr>
<tr>
<td>SNMP (simple)</td>
<td>110.1</td>
<td>320.7</td>
<td></td>
</tr>
<tr>
<td>STARC (complex)</td>
<td>795.3</td>
<td>799.0</td>
<td>878.8</td>
</tr>
<tr>
<td>SNMP (complex)</td>
<td>165.5</td>
<td>610.7</td>
<td></td>
</tr>
</tbody>
</table>

in a central repository, but managed and created by one trusted entity. This case is no worst that the use of locally installed probing classes. In the second scenario, middleware users create probing classes to be uploaded. In this case, it is necessary to execute them in a restricted environment, such as virtual machine, similar to where the scheduled jobs would be deployed.

In order to guarantee a timely response from the probing classes and to prevent denial of service attacks, timeouts are enforced on probing class execution (returning 0.0).

V. Evaluation

In this section we describe the evaluation of STARC resorting to qualitative analysis and micro-benchmark performance results. In qualitative terms, STARC provides a number of advantages w.r.t. related work. It allows a more expressive, flexible description of resource requirements for jobs. It encompasses the notions of user-specified ranges of (non-linear) partial-utility, and provides different policies to evaluate combined utility for complex requirements (priority, strict, balanced, elastic). Such policies can be selected by users or enforced by the system providing different levels of quality-of-service. Existing systems rely on: i) binary decision on requirement fulfillment (as service-level agreements) not considering partial utility, or ii) adopt solely a system-centric approach by employing complex, predefined (not user-specified) utility functions in order to optimize request scheduling based on budget, deadlines, or iii) combination of both.

Regarding performance, we designed a micro-benchmark to evaluate the overhead brought to every host when evaluating a XML requirement file. The measurements consider a LAN setting as a worst-case scenario where the relative overhead of requirement evaluation w.r.t. communication is the highest. In order to measure the requirement evaluation times we used a 3.2 GHz Pentium 4 personal computer with 1 Gb of memory as a resource provider where all resource evaluations were made. The client computers, where all remote evaluations were initiated are Apple iBook with a 800MHz PowerPC processor, 640Mb of memory and the Linux operating system. All computers were connected by a 100Mb switched Ethernet network.

The micro-benchmark measurements comprise timing the evaluation of two sample requirements locally and at remote hosts: i) a very simple requirement, merely stating a minimum necessary amount of memory; and ii) a complex requirement which is a conjunction of 20 simple requirements. These measurements take into account the need to initially upload code for Environment Probing Classes and are then compared to two network protocols for resource evaluation that serve as yardsticks: i) WBEM [27] used to measure memory locally available at each host, and ii) using SNMP [33] programmatically to evaluate resources at each host using `GetRequest` and `GetBulkRequest` methods. The Local experiments were performed accessing directly the data source, while the STARC experiments used the proposed platform. The results are depicted in Table II.

The results allow to conclude that STARC scales w.r.t. the size of the requirement files (simple vs. complex) and regarding local and remote evaluation. Uploading the memory evaluation code takes only 80 milliseconds. Naturally, other Environment Probing Classes will take different times to upload. When evaluating simple requirements, STARC adds a small overhead to WBEM and performs better than SNMP. In the case of the complex requirements, using the SNMP API is better than STARC; however the evaluation of the logical operators must be explicitly programmed.

![Figure 6. Concurrent requirements evaluation overload](image)

To evaluate how STARC behaves under load, we measured the time to evaluate series of complex requirements concurrently. These results are shown on Fig. 6.

The graph clearly shows that until about 100 concurrent
evaluations there is no performance degradation. Furthermore, the interleaving of the various processes evaluating requirement policies leads to a reduction of the average processing time. This is naturally due to the operating system being able to better utilize the CPU, by executing some threads while others wait for I/O due to virtual memory.

From that point onwards (about 100 concurrent evaluations) the scheduling and process swapping costs are greater than the gains from the interleaving of the parallel execution. Nonetheless, the middleware maintains scalable behavior since with 200 concurrent processes performing complex policy evaluations, there is only an increase of about 10% on the average requirement evaluation time. To access the WBEM service a Pyro RPC server was implemented, which has a limit of 200 active network connections, hence the maximum value of concurrent policies tested. We can conclude that requirement evaluation in STARC scales well to large numbers of concurrent clients based on the same conclusions of previous studies [14].

By leveraging a P2P infrastructure [32], STARC can also scale to large numbers of participating nodes (peers).

VI. CONCLUSIONS

STARC is capable of evaluating and comparing different resource providers with respect to client specific resource requirements, considering ranges of partial fulfillment and utility depreciation. We are able to evaluate a set of standard resources in different computer architectures: number and speed of CPU cores, available memory, available disk space, network speed, among others, according to different client profiles.

With the proposed utility algebra and corresponding XML DTD it is possible to define any kind of requirement a module or a complete application may have and that a resource provider must satisfy. The use of logic operators eases the comparison of different hosts. By using these operators the result of the evaluation returns a numeric value that clearly states how the requirements are totally or partially fulfilled. The values returned by all hosts are easily compared to find the one(s) that best fit the client requests, w.r.t. all required resources and perceived utility depreciation (partial-utility), according to a combined evaluation policy. This way, resource discovery is more effective (and will in fewer resource discovery failures) than a simple matching approach.

The architecture of this system allows its extensibility by allowing the definition of new types of resources to be discovered and evaluated. The code to evaluate a resource can be dynamically installed, without system compilation, and reused on behalf of many clients.

A. Future Work

We are currently implementing a STARC module to deploy on MDS4 [5] integrating the utility algebra in MDS4 Aggregator Framework. In the future, we plan to use the STARC in a scenario with job migration, where it is also necessary to adapt to situations where available resources in a resource provider change during job execution. Since periodically polling all hosts is expensive, a possibility is to register in the remote hosts the threshold or change rate above which the client wants to be notified. The middleware architecture we propose easily allows the implementation of such alternatives.

Some resources can not be easily quantified, hence we plan to allow the writing of fuzzy set membership expressions close to natural language (is large, is small, is good, requires, etc.) to ease the specification of the requirements, in a similar way as the offloading rules presented by Xiaohui Gu et al. [34]. It will also be necessary to evaluate how these expressions and other fuzzy logic operators (Yager or probability product/sum) are close to the perception a user has of a computer processing power.

REFERENCES


