The Computer Journal Advance Access published March 2, 2015

doi:10.1093/comjnl/bxv012

C^3P: A Re-Configurable Framework to Design *Cycle-sharing Computing Cloud Platforms*

Sérgio Esteves*, Paulo Ferreira and Luís Veiga

INESC-ID/Instituto Superior Técnico, Universidade de Lisboa, Lisboa, Portugal
**Corresponding author: sesteves@gsd.inesc-id.pt*

A new era of High-Performance Computing has been coming about during the last decade. The overabundance of resources lying idle throughout the Internet, for long periods of time, calls for resource-sharing infrastructures operating in the settings of the Cluster, Grid, P2P and Cloud. Many organizations own grids, frequently underutilized, but impose several restrictions to their usage by outside users. Despite the already extensive study in the field of Grid and Cloud computing, no solution was ever successful in reaching out to typical home users and their resource-intensive commodity applications. This is especially lacking in an open environment with no cost and low access barriers (e.g. authentication, configuration). We propose C^{3}P, a comprehensive distributed cycle-sharing framework for enabling the sharing of computational resources in a decentralized and free computing cloud platform, across large-scale networks and thus improve the performance of commonly used applications. C^{3}P encompasses the following activities: application adaptation, job scheduling, resource discovery, reliability of job results and overlay network management. C^{3}P evaluation shows that any ordinary Internet user is able to easily and effectively take advantage of remote resources, namely CPU cycles, for their own benefit; or provide spare cycles to other users, getting incentives in return, in a free, yet fair and managed global infrastructure.

Keywords: grid; peer-to-peer; cycle sharing; cloud computing; public computing; middleware frameworks

Received 27 September 2013; revised 24 January 2015
Handling editor: Fionn Murtagh

1. INTRODUCTION

During the last decade and a half, high-performance, high-throughput and large-scale computing in general have attracted significant interest from the scientific community. Computers and networks with increasing capabilities at lowering costs, along with Internet wide access, have been regarded as one of the most viable solutions to the unmistakably increasing needs of solving today's science problems. These demand greater and greater computational resources (processing, storage, bandwidth). In fact, many sciences, ranging from natural sciences (e.g. physics, chemistry, biology) to social sciences and humanities (e.g. economics, sociology with social networks, linguistics) have become, to a large extent, e-Sciences, in the sense that they rely on heavy computations to discover, produce, process and validate or disprove most of their scientific findings. Thus far, computer resources have been harvested relying on well-known models, such as the cluster, the grid, peer-to-peer (P2P) and, lately, the cloud. Additionally, many contents related to the analysis and presentation of e-Science results rely more and more on multimedia that is heavy to generate, process and present, namely so in medical and life sciences, physics, astronomy, chemistry, materials, mechanics and networking.

Regarding Internet home users, there is also need for increasing computational resources in activities carried out by enthusiastic amateurs or hobbyists. Examples include the use of applications to render photo-realistic images and animations using ray tracing, video transcoding for format conversion, batch picture processing for photo enhancement, face detection and identification, among others. Many of such applications are free or shareware, and widely available and deployed across the Internet.

Shortcomings of current solutions: Most infrastructures owned by companies and research institutions may encompass

dedicated data-centers, clusters or desktop computers, all federated and leveraged for cycle-sharing scheduling as they are frequently underutilized. This gives place to almost inexpensive computing resources available that, being frequently extensive, need to be harnessed, governed and made available to users' applications in an easy and efficient manner. In the settings of cluster, grid, P2P and cloud computing, this is handled by resource discovery (RD); scheduling and management and middleware (i.e. application programming interfaces, application development, deployment and adaptation).

While there are many solutions in the literature to address this, none of them has been successful in the goal of bringing these settings to the typical non-expert ordinary Internet home users, many of whom nowadays already have significant computation needs, usually related to multimedia content generation and processing. To the best of our knowledge, there is no proposed solution in cycle-sharing-related technical literature that has been successful in the deployment of the overall Grid paradigm on top of large-scale networks with no cost or low access barriers, and suiting any ordinary user and their unmodified CPU-intensive applications.

Usual restrictions and shortcomings often include aspects regarding: authentication, difficult configurations, inflexibility of master-slave models; absence of a useful set of applications targeted; users being able to only donate cycles (typical in client–server models); authorization and authentication models precluding the typical computer user (typical in institutional Grids); being directed mostly for science and engineering, forgetting desktop applications; being not easy to use; configurations being hard to manage; jobs being platform dependent; idleness levels being disregarded; applications needing to be modified (data-based parallelism not exploited).

Proposed contribution: Given the current context, we propose a comprehensive middleware platform that eases the sharing of computational resources across large-scale networks. Better still, we intend to provide open access where any non-expert user may consume remote resources from other machines or provide their own idle resources to others. Such a platform should be designed as a global re-configurable and adaptable meta-architecture capable of functioning in environments from the simple cluster to the Cloud.

The solution here presented is named C^{3}P (Middleware Framework for *Cycle-sharing Computing Cloud Platforms*),[1] and it is designed around two main guiding principles:

(1) The notion of ***gridlet***, an abstraction of a chunk of data with meta-data (or code). It is used consistently a semantics-aware unit of work, in the context of RD and management, workload division and task deployment, accounting and reputation, computation off-load and application adaptation. Gridlets serve to C^{3}P the same purpose of a unifying common minimal interface, as TCP packets in many network and distributed application protocols.

(2) A layered, composable, component-based middleware architectural framework, where the relevant concerns and interactions within a cycle-sharing platform are identified and abstracted. This frees the developer of middleware components encasing each specific mechanism, from the details related to the underlying sources of computing resources and deployment infrastructures, such as clusters, grids, P2P overlay networks and cloud infrastructures. Moreover, it allows easy creation of a tailored cycle-sharing platform targeting a specific scenario by declaratively defining a configuration of middleware components.

The metaphor, approach and architectural framework, together with the detailed implemented components, provide the following novelties regarding previous work pursuing the same goals:

(i) novel mechanisms to discover, allocate and schedule resources for cycle-sharing systems running on large-scale networks, such as P2P grids and community clouds;
(ii) novel data-driven and -aware model and methodology for transparent parallelization of common desktop applications, without requiring the modification of the original source code or even binaries of the application;
(iii) novel mechanisms to verify results and improve reliability of executions on large-scale systems, in the presence of malicious and free-rider nodes;
(iv) significant out-of-the-box performance speedups for CPU-intensive applications, without the need to develop according to specific APIs or Grid middleware;
(v) additional flexibility for cycle-sharing architecture design itself, as the middleware framework can cope to a significant extent with different sets of components and mechanisms implemented by those components.

Overall, an important advantage of our solution is that it enables the transparent parallelization of common desktop applications without the need to introduce changes in their source code. With the gridlet metaphor, we rely on a data-based parallelism in which the input files and parameters, of an application are divided and encapsulated into gridlets to be executed on remote machines. To exercise these design principles, we developed a solution bundled with components already implementing the more relevant aspects, encompassing the following functionalities: application adaptation mechanisms, gridlet creation (GC) and scheduling, network management, reliability mechanisms, RD, accounting and incentives.

System administrators may configure a composition of components (or reuse a pre-built one) in order to deploy a

[1] We use the C^{3}P acronym to avoid collision with *CCCP* that stands as a country acronym in the Cyrillic alphabet.

custom middleware platform targeting the intended scenario. Middleware developers may target one of these aspects (e.g. accounting integration with a PayPal or institutional credit system module), develop a new component, make it available to be plugged in configurations, without having to worry about how other aspects (e.g. scheduling) are handled. Users and application developers/adapters should be exempted from platform details.

Document road-map: This article is organized as follows. In the next section, we present the core of the C^{3}P middleware framework, inspired by the gridlet metaphor, with the components responsible for providing the enabling mechanisms of a archetypal cycle-sharing platform. We also highlight specifics of C^{3}P deployment on most popular settings. In Section 3, we offer relevant implementation details, regarding the core organization of the framework and a number of representative components. Section 4 describes the experimental setup and results to demonstrate the qualitative aspects and assess the quantitative metrics of C^{3}P performance. Work found in the literature related with C^{3}P is discussed in Section 5. The paper closes with some conclusions in Section 6.

2. C^{3}P MODEL AND ARCHITECTURE

In this section, we describe the model, architecture and design choices of C^{3}P. C^{3}P is a framework addressing the development and deployment of distributed resource-sharing middleware platforms for large-scale, high-performance and high-throughput computing, with multiple paradigms targeting different settings. It is mainly dedicated to enable users to share computer resources across a network, local or wide areas, to exploit parallel execution of commonly used applications and, hence, improve their performance. This, in an easily extendable and adaptable way.

The case for an integrated cluster, grid, P2P and cloud ecosystem: Cluster computing has enabled the parallel execution of applications running on multiple computers connected via a local network. In many ways, accessing the Cluster is like accessing a single computer, with the advantage of being cheaper than traditional supercomputers and more easily scalable. The message passing interface is one of the most relevant paradigms addressing the communication of data in parallel applications running on the Cluster.

Grid computing provides an aggregation of networked loosely coupled computers (or clusters) that can be heterogeneous and geographically dispersed. Normally, this model is for internal use by a single organization/institution or consortium (business, university, virtual organization, etc.), imposing heavy authentication and restrictions over the sharing of resources. Also, it allows the federation of multiple clusters under the same administrative control, taking advantage of a wider range of computational resources that might be otherwise underused.

Since its inception, P2P architectures have been gaining considerable research attention and widespread use across the Internet. P2P achieves higher scalability by allowing a much greater number of machines connected and, unlike the Grid, achieves higher decentralization, since it does not require the intermediation of centralized servers or authorities, which could suffer from typical problems such as bottlenecks or single points of failure.

Cloud computing provides a useful layer of abstraction over the networked-linked resources. The main idea is that a developer should only focus on the application he wants to build, and not on the underlying infrastructure used to scale that same application. Resources are dynamically allocated in accordance with the application's needs and, in case of public clouds, a customer has only to pay for what he gets, generating the unprecedented utility computing paradigm.

These models have been designed over time as attempts to improve application performance, scalability as well resource utilization (effective usage of resources made available) and scheduling (fairness or balance in resource allocation). Such models have been combined with one another so that their best properties may be joined together. For example, a grid P2P infrastructure allows the mutual sharing of resources in a higher scale. Nonetheless, the bottom line is that all of these models are aggregations of computers, connected over a network, that might be idle for long periods of time and can be used to compute resource-intensive applications in parallel.

Nowadays, the scarce resource utilization of already powerful computers lying all over the Internet is creating a great opportunity for these computing models to take place. In the matter of content distribution over the web, P2P protocols have already been widely exploited. As for cycle-sharing, several barriers still stand, such as the parallelization of common desktop applications.

We envision that not only scientists and engineers, but also typical Internet users and general society could enjoy augmented performance on their commodity applications, by adding remote resources to their computations.

Gridlets: As already described, in C^{3}P, the notion of gridlet is central (Fig. 1 depicts the general architecture overview). It is a semantics-aware unit of workload division and computation off-load. A gridlet is a chunk of data associated with the operations to be performed on the data, and in many cases these operations consist of unmodified application binaries. In C^{3}P, gridlets are the basic units of processing scheduling, data transfer, RD and accounting of contribution to the system. We assume the existence of a standard gridlet (e.g. derived from the Linpack benchmark or computational puzzle), used as a reference unit for comparison, and to correctly take into account the different capability of nodes.

Without loss of further detail in this and in the next section, each gridlet goes through a life cycle (Fig. 2) and has an associated cost (that may be an estimate or an actual value),

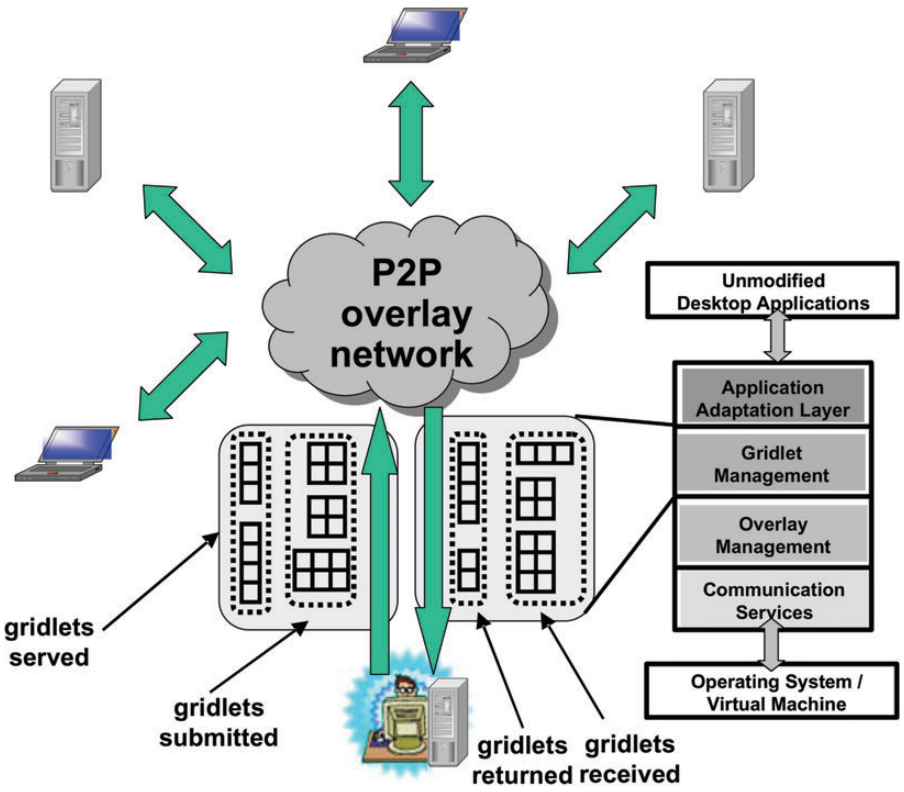

FIGURE 1. C^{3}P network architecture overview.

which will take into account the processing, bandwidth and memory necessary to execute it. The gridlet application model essentially divides application execution in the following phases: (1) GC, (2) gridlet processing and (3) gridlet-result aggregation (GRA). Gridlet processing can be further divided into: (2a) gridlet data injection, (2b) application execution and (2c) gridlet-result extraction. Phase 1 preprocesses and splits input and triggers RD so that Phase 2 can take place. Phases (2a) and (2b) perform application adaptation (via data and parameters, more details later) so that an unmodified binary may be executed in (2b). Phase 3 is analogous to Phase 1, yet carried out in a converse manner.

2.1. C^{3}P framework

The C^{3}P framework follows an archetypal architecture (or meta-architecture) for a cycle-sharing middleware, described in Fig. 3. It comprises four layers: the application adaptation layer, which encapsulates specific mechanisms to handle each supported application; gridlet management and reliability, which handles gridlet processing; network and resource management, which is responsible for the operations finding resources and allocating them and data transfers over the network (e.g. a P2P overlay) and Communication Service, which abstracts the means for actual distributed interaction, with addressing, routing and sending messages (much as a middleware counterpart to the MAC layer, it could encapsulate Java RMI, Python or C sockets).

The four layers comprise a number of components that address the various required mechanisms we found in cycle-sharing (or Internet distributed computing, public computing) platforms throughout the literature, in here, defined around gridlets. Each layer has a core component (gray) as any system, however minimalist, provides such a mechanism (e.g. an application adaptation could be an API library, scheduling could be an FIFO queue, network management may be a simple list of known hosts and message routing (MR) may resort to ftp across hosts). In most settings, however, one or more of these layers must have additional mechanisms or functionality in a given mechanism, such as briefly introduced in the next paragraphs. Finally, some mechanisms such as those for reliability and replicated storage are not essential (in functional terms) and thus non-mandatory (green).

Application adaptation engine (*AAE*): This component handles application-dependent mechanisms that are described through an XML-based declarative specification language. It gives semantics to the described rules in order to provide transparency to the processes of configuring and parameterizing applications, and splitting and merging data.

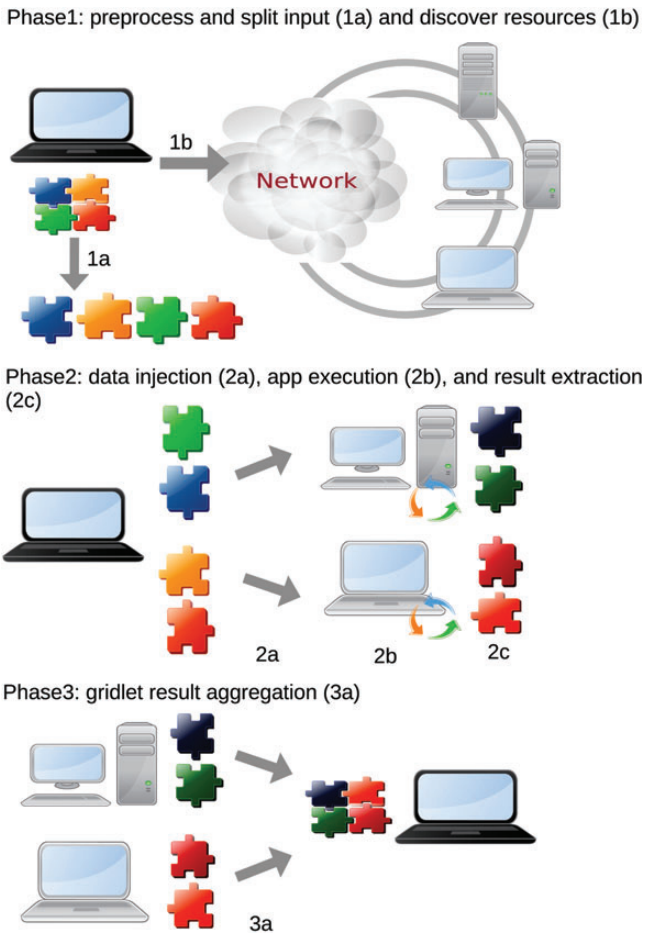

FIGURE 2. Gridlet life cycle.

Execution environment (*EE*): To provide a safe environment to execute applications with external input data, this component provides a sandbox and virtualization technology (system VM or managed language VM). It can also allow interaction with job submission middleware (clusters, grids), workflow engines or cloud infrastructures.

Gridlet scheduler (*GS*): This component plays a major role in C^3P as it coordinates all the interactions between all the other high-level components. From some GUI or command line, the GS keeps waiting for commands to initiate jobs. It is responsible for maintaining the whole state of the jobs; asking other components on the same layer to perform specific tasks, like creating or aggregating gridlets and interacting with the layer below, namely to send or accept gridlets.

Gridlet creation: This component creates properly formed gridlets and assesses their cost against the standard gridlet. To split input data or generate different configuration for each gridlet, the GM uses the Application Adaptation Layer.

Gridlet-result aggregation: The purpose of this component is to merge and store gridlet results and build the application's final output when a job is completed. The mechanism of joining gridlets and building application output is made in accordance with the rules defined for that application or data format in the AAE.

Gridlet-result verification (*GRV*): In a consumer node, this component is responsible for validating the received gridlet results by checking overlapping chunks of redundant data. Gridlets whose results are considered as not valid have to be rescheduled by the GS, and the provider nodes that produced the wrong results have to be discarded from the new set of candidate nodes to process the failed gridlets.

Currency and reputation (*CR*): This component is used, normally by a subset of nodes (e.g. a central server or super-peers in an overlay), in order to keep a distributed reputation system for all the nodes. For instance, in a P2P overlay, each super-peer is responsible to keep track of its direct children nodes in terms of reputation and currency. The details of reputation and currency may be hidden from the other components and are described in detail later.

Network manager (*NM*): This component coordinates all the interactions between the components on this layer. It is responsible for the operations of routing and addressing over the network. From the above layer, this component receives gridlets and distributes them to other nodes through the layer below. Conversely, it delivers gridlets coming from the layer below to the above one to be processed.

Resource manager (*RM*): The purpose of this component is to assess the capability and availability levels of the local resources on the machine (CPU, memory and bandwidth). This is done through system calls to the operating system or resorting to well-known benchmarks, in the case of CPU.

RD: This component provides mechanisms for locating computational resources within the network. The NM calls this component specifying parameters, like minimum capability and availability required, and it returns a set of nodes that match those requirements. Furthermore, there are usually two levels considered: using the resources of sibling or neighbor nodes directly; and, if required, using relay across the network (such as via super-nodes) to reach other nodes.

Gridlet-result storage (*GRS*): To avoid redundant execution of gridlets, this component has the purpose of storing gridlet results within the network. The GS will look up for gridlet results here soon after the GC process. Also, when a gridlet result is deemed trustworthy, it is sent to this component to be persisted, or to allow decoupling in time between consumers and producers.

MR: This component handles all of the low-level mechanisms related to sending and receiving messages to and from the network, such as translating node identifiers in hostname/-port pairs. The NM uses this component to send messages to the other nodes in the network. Also, whenever a message from the network is received, this component analyzes the message in the first instance and then delivers it to the adequate handler routine in the NM.

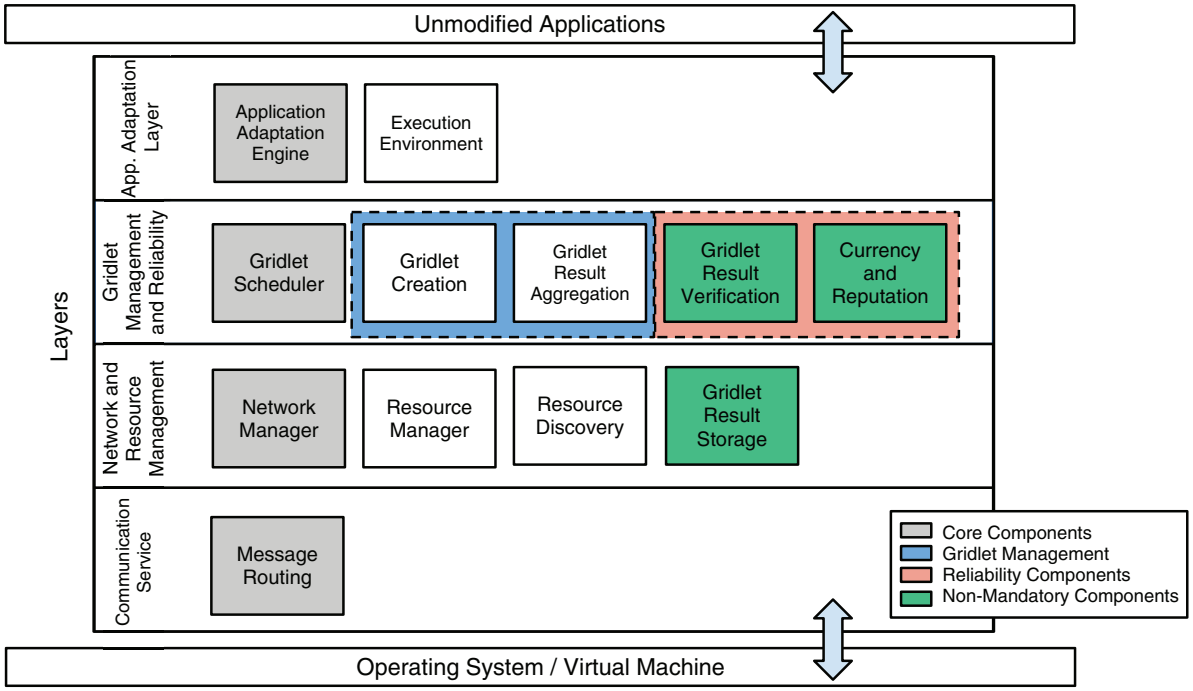

FIGURE 3. C^{3}P archetypal layer and component architecture.

We recall that not all of these components are mandatory to be included in a cycle-sharing middleware platform. For example, the reliability components may be discarded if the platform is running on a trusted environment, such as an institutional grid. In the same way, the GRS can be precluded if the data in executed gridlets are very different from one another. Nonetheless, the components colored in gray, the core components, cannot be removed as they control the entire workflow of the platform.

In order to not overload a single machine, some components may run in different computers, especially the component that launches applications to perform external gridlets, EE, as it could be very resource-intensive. Moreover, it is possible to have multiple EEs in various machines controlled by the same node. In this case, the machine that runs the core components must implement additional code to call the other remote components, e.g. through remote procedure calls (RPCs) or web services.

Interactions between layers: The core components must be implemented according to an API (addressed in the next section) so that each layer can provide/invoke the following functionality to/from other layers:

(i) The Gridlet Management and Reliability needs the following functions from the Application Adaptation Layer: (i) launch applications with the correct parameters and input files; (ii) launch applications to execute part of a gridlet (e.g. a single chunk) to calculate its cost; (iii) split input data to build gridlets; (iv) merge output data to build the application final output.

(ii) The Gridlet Management and Reliability needs the following functions from the Network and Resource Management: (i) submit gridlets to be processed by remote nodes; (ii) return gridlet result to the consumer node; (iii) look up gridlet result in distributed storage; (iv) save gridlet result in distributed storage; (v) update resource availability levels; (vi) get candidate nodes to process gridlets.

(iii) The Gridlet Management and Reliability needs to provide the following functions so that it can be called by some user interface: (i) process work; (ii) return progress indication; (iii) register, update and remove application with format descriptors.

(iv) Network and Resource Management needs the Gridlet Management and Reliability to: (i) deliver gridlet result; (ii) deliver gridlet to be processed.

(v) Network and Resource Management needs the Communication Service to: (i) route messages to some address.

(vi) Communication Service needs Network and Resource Management to: (i) deliver message. Each API can be implemented differently (the approach used to tackle different settings) and can also be extended. Component manifests and reflection are used to expose such extensions to other components, but their semantics must be known by middleware developers of other components in order to be leveraged.

2.2. Prototypical setting: a P2P Grid

Without loss of generality, we now consider how C^{3}P can be instantiated in a large-scale scenario such as a P2P cycle-sharing system or P2P Grid. As the first instance of our network configuration, we consider a C^{3}P grid-overlay consisting of an hybrid P2P overlay, where each node can act as a resource consumer, provider or, additionally, as a super-peer, performing special functions within the overlay. Super-peers form a ring (such as Pastry [1] or Chord [2]) amongst themselves (or optionally, are organized in a CAN [3]) and they aggregate information about their child nodes, namely application indexing and node reputation maintenance/accounting.

The C^{3}P middleware runs on each node enrolled in a C^{3}P grid-overlay, and follows a vertical layered architecture to favor portability and extensibility (Fig. 3). In this particular work, all jobs are composed of independent, non-communicating tasks/gridlets, commonly referred to as embarrassingly parallel tasks.

Super-Peers: Figure 4 depicts the P2P overlay organization. Super-peers form a ring among themselves to allow faster communication and avoid hopping messages through the overlay network. They share information about the availability of applications among their child nodes and act as resource brokers, sharing their child nodes' resources with each other when such is needed (for instance, when a super-peer's children cannot perform the work required for a given gridlet, it requests work from another super-peer whose children have enough availability to perform the task). Regular nodes, in turn, are 'clustered' around these super-peers and use them to perform service discovery and forward gridlet requests to other nodes in the network. Every node is assigned to a single super-peer, determined from all known super-peers through the Pastry proximity metric; it is then referred to as that node's primary super-peer. The use of the proximity metric was chosen because it explicitly represents the degree of proximity between two given nodes in the overlay, as defined by the basic Pastry protocol.

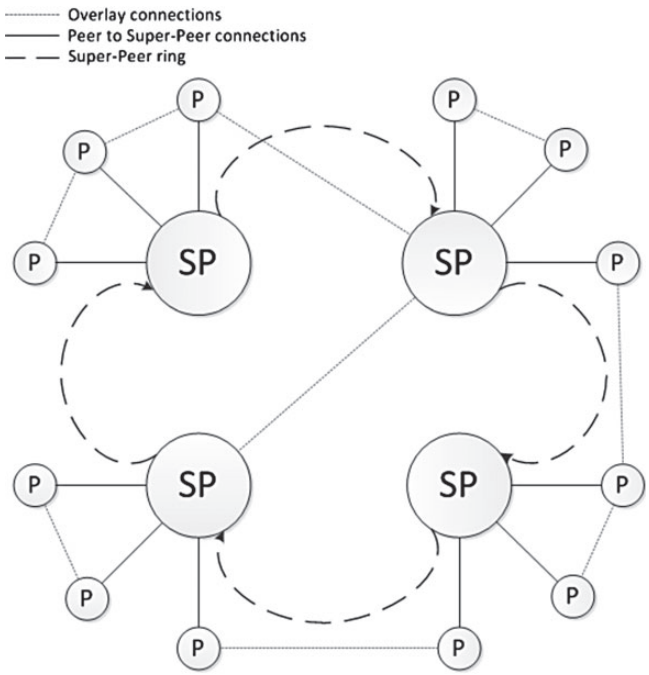

FIGURE 4. Node organization in the network. *SP* indicates a super-peer, while *P* indicates a regular peer

Despite being assigned to a single super-peer, every node possesses a list of existing super-peers in order to choose a new super-peer in case of failure of the existing one. Election of super-peers is performed in a way that attempts to balance the ratio of super-peers to regular nodes and make sure no nodes are orphaned, i.e. left without knowing any super-peers. Nodes will periodically check the number of super-peers they know; if this number is below a certain threshold, those nodes can select themselves to become super-peers with a given probability. This probability has to be kept low enough so that the number of super-peers cannot grow too large at any given point in time, but such that allows a fairly rapid expansion of the number of super-peers in case of shortage.

Resource access model and gridlet life cycle: We now describe the gridlet life cycle and resulting resource access model. First, a user specifies the application, parameters and input files through a user interface. The GS then creates gridlets from the given input data and calculates their cost, e.g. by executing some samples and comparing them to the standard gridlet.

Then, after a minimum amount of time, for each gridlet, the GS verifies if its result is available in the GRS. If so, the result is retrieved and forwarded to the GRA component. Otherwise, gridlets are forwarded to the NM which, by its turn, tries to gather a set of candidate nodes that are available to process work.

The NM locates resources firstly by looking up its internal cache for neighbor nodes, and, if required, contacting its primary super-peer to reach other nodes in the network. Soon after, the gridlets are distributed through the candidate provider nodes (this completes Phase (1)). For reliability purposes, each gridlet should be allocated to k different nodes, where k is a replication factor.

Whenever a resource provider node receives a gridlet, the MR sends it to the adequate handler routine in the NM. In turn, the NM sends the gridlet to the GS, which executes the gridlet by launching the respective application with the correct parameters and input files, possibly subject to adaptation (e.g. parameter transformation and/or data wrapping/un-wrapping). Upon gridlet completion, its result is sent to the resource consumer node through the NM and MR components (Phase 2).

The resource consumer receives the gridlet result and sends it upwards to the GS, which, by turn, sends the data to the GRV. If a k replication factor of results of a same gridlet exists inside the GRV, the results are validated against each other. If a majority of results is accepted, one of the valid results is sent both to the GRA and the GRV, and the primary super-peer of the consumer is informed about which results were accepted and from which provider nodes, so that their reputation and currency may be increased (Phase 3). Otherwise, the corresponding gridlet has to be rescheduled and the super-peer has to decrease the reputation of the nodes that produced the wrong results.

When the GRA contains all gridlet results of a same job, the final output of the application is built with the AAE and a notification is sent to the user interface.

In the following subsections, we describe the design (algorithms, protocol and data structuring) of the more relevant mechanisms, covering from application adaption to basic RD across the network.

2.3. Application adaptation and gridlet management

Gridlets are small work units that should be distributed in a proportional manner, for performance reasons, to nodes with different capabilities. As an example, if there are three available provider nodes to process a job, two of them having half the capabilities of the other one, then the more powerful provider would receive 50% of the gridlets and the other two 25% each. This promotes an ideal situation where all gridlets should have the same cost, or as approximate as possible.

The cost of a gridlet can be parameterized and should be small both in size and computational complexity, for easy relaying and restarting. This requires the assessment of the cost of the whole input, which is done by computing small random parts of the job. For example, if we have a job concerning an image to be rendered with 100 chunks, then we may compute three random chunks and determine the average cost of a single chunk in terms of consumed resources (elapsed time, memory taken and transfer size). Knowing this allows efficient and balanced distribution of chunks into gridlets.

The size of gridlets in a job is mostly fixed for rescheduling, caching and pipelining purposes. First, sending heavy gridlets to powerful nodes might not be a good idea, as machines may fail while performing gridlets (i.e. gridlets would need to be rescheduled to other nodes that might not be so powerful). Secondly, having different-sized gridlets for the same job would hinder the search for gridlet results already computed in some cache. Finally, a node can be processing a gridlet while receiving others, as a great part of the difficulty of achieving high performance comes from the need to orchestrate communication and computation.

With regard to the parallelization of applications, they need not be modified to work with the C^{3}P framework. This significantly lowers the access barrier of home users into distributed cycle-sharing systems, as either they do not have to possess any expertise in software development or the applications they use are closed-source or proprietary. Instead, we rely on a data-based parallelism in which the input data of a job is decomposed into smaller units (gridlets) with associated tasks to be distributed and executed in cycle-sharing machines. Later, the output data generated by the provider machines are reassembled into a single result.

In C^{3}P, application-dependent mechanisms are handled transparently and automatically by the AAE component, and it only requires access to an available format description of the application input and output. Better still, this format description is an XML-driven grammar, written in a high-level language, containing all the necessary rules to split input data and merge output data. This includes header analysis and reconstruction, patching and structural modifications such as moving blocks across the file. This is addressed in detail in Section 3.2 and in Listing 1. Furthermore, format descriptions for widely used applications may be obtained transparently through an online repository, or neighbor nodes.

2.4. Gridlet-result verification

The results returned by provider nodes may be wrong due either to machine failures or malicious behavior. Malicious nodes create fake results that are intentionally difficult to detect. Their motivation is to discredit the system, or raise a reputation for work they have not performed.

To improve reliability against wrong or forged gridlet results, every given gridlet should be executed in k different nodes, where k is a replication factor. This k may vary with the reputation of provider nodes that will execute the gridlet. For example, if the first node receiving a gridlet has a reputation above a given threshold considered trustworthy, then k may be reduced. In C^{3}P, this is obtained through the following equation:

$$k = \begin{cases} max_k, & \text{if } reput < threshold, \\ \dfrac{reput - threshold}{(max_{reput} - threshold)/max_k)}, & \text{otherwise.} \end{cases}$$

After gridlet computation, instead of accepting results based on a voting quorum scheme (that can be easily fooled if groups of colluding nodes decide to return the same wrong results), we propose another approach, as described next.

Samplication: The replication model might not be sufficient as the majority of gridlet results are provided by untrusted third parties. As such, we considered to add a sampling model, consisting in the local execution of a fragment, as small as possible, of a gridlet to be compared with its returned results.

In C^{3}P, we use replication and sampling sequentially to achieve higher reliability of the results. The samples are chosen from mismatch areas of the replicated results or randomly if there is no mismatch. Results that do not match the sample

are promptly discarded. Moreover, we propose the following algorithm for *samplication* in the listing below.

```
1  schedule redundant work, put the results in a bag;
2  if bag is empty then
3  |  goto(1);
4  end
5  if all results in the bag are equal then
6  |  if random sample matches then
7  |  |  accept result;
8  |  else
9  |  |  remove all results from the bag;
10 |  |  goto(1);
11 |  end
12 else
13 |  choose a sample within the mismatch area;
14 |  compare with all results;
15 |  remove results from the bag that mismatch the
   |  sample;
16 |  goto(2);
17 end
```

Furthermore, this scheme can still be tricked, albeit not so easily as the voting quorum scheme, if colluding nodes submit equal gridlet results in which some fragments (but not all) are forged, and the randomly taken samples refer to other fragments (that are correct). This is less likely to happen with higher number of samples, and thus there must be a compromise between this number and the incurred overhead of execution.

2.5. Currency and reputation

C^{3}P incorporates the concept of resource usage fairness. The nodes contributing more to the community, by providing more access time to their spare resources, should also be capable of assembling more available resources for their own, when they need to execute gridlets. On the other hand, nodes contributing less to the community, by providing less resources or denying access to them, should have their priority decreased when they need to execute gridlets.

This exchange of resources is controlled by a currency economy where, unlike a token economy, resources may have different values based on their capability and availability levels. For example, a machine with 60 Mbps of available upstream bandwidth would be more expensive than a machine with 24 Mbps.

We also consider that resource prices should vary with the demand and supply like in real-world businesses. If some resource is scarce within the network, due to great demand in 'rush hours' for instance, then its price should be increased; or, inversely, if there is an overabundance of some resource, its price should be decreased. In this way, the overall throughput of the system—number of processed gridlets in a given time—tends to be higher and the collaboration between users is fostered.

In an open environment, where malicious users might reside, knowing which nodes are trusted is crucial for preventing faked/forged results. C^{3}P provides a reputation system where nodes can attribute reputation to each other. The reputation of an entity can be described as the result of the level of trust peers place on that entity. In our case, the reputation of a node is simply the quotient between the number of valid returned results and the number of total executed gridlets.

We use reputation when a consumer node needs to locate and select a set of nodes for performing a job. Only nodes with a reputation above a specified threshold on the consumer can be selected, and, within them, the most well-reputed ones are preferred. The reputation system can also be consulted by well-reputed providers to reject work from consumers with low reputation. This gives an incentive for nodes not to forge results. Finally, reputation can help in reducing gridlet redundancy, as previously described (Section 2.4).

Usually, a central entity or a subset of nodes, such as super-peers is responsible for maintaining the reputation values of their direct child nodes. These values are periodically adjusted downward to promote activity; both the time interval and reputation points are configurable.

2.6. Resource management and discovery

In the context of a distributed cycle-sharing platform, regardless of it being an idle cluster, institutional grid, P2P grid or cloud-like infrastructure, the resources considered are CPU, primary and secondary memory and bandwidth. Resource availability is assessed every time changes in resource utilization occur, i.e. on node's bootstrapping, before and after gridlet computation and within a specified time frame, since availability may change due to external process activity. This last option may be always used in order to reduce network traffic.

To measure CPU availability, we rely on the Linpack benchmark [4] where a system of linear equations is solved. In this way, we get the execution time or GFLOPS taken, no mattering if machines have different CPU architectures. Regarding primary and secondary memory, they are simply assessed through operating system calls. Finally, our approach to measure bandwidth relies on checking the length of time for a message to travel from one node to another and back again. This is done by taking advantage of regular messages (gridlets) of the system already exchanged between a node and their neighbors. Within a short period of time, the minimum round-trip time obtained, among all exchanged messages with neighbors, is kept and the bandwidth is trivially calculated. In this way, we may find the useful bandwidth between a node and its neighbor with highest bandwidth. If the neighbor has more upstream and downstream bandwidth than the node, then we have found the node's bandwidth.

Nonetheless, installed applications and services in nodes may also be regarded as (software) resources. The super-nodes in the C^{3}P grid-overlay are responsible for indexing applications, thereby keeping information about the family hash and application hash. The family hash is a hash of the canonical name of an application, for example, the URL of the application's main web page online (e.g. www.ffmpeg.org for the ffmpeg application).

In the super-nodes, this family hash is used to aggregate information about the availability levels of nodes that have applications that fall in that family. More precisely, for each family, super-nodes maintain a table containing both the capability of all of their child nodes and the aggregated capability of nodes belonging to the domain of other (possibly a fraction or neighboring) super-nodes. If a super-node ID is larger and numerically closest to the key given by the family hash, then the super-node is responsible for that family. This is done in order to evenly distribute the application family's key space among all super-nodes and avoid overloading any given one.

Concerning the application hash, it represents a hash of the application name, version and, if needed, operating system. It is used inside a super-node, responsible for the respective application family, to find which nodes have a specific application or version installed. For example, considering the ffmpeg application family, there could be a number of distinct application versions described uniquely by this hash such as ffmpeg 0.6 and ffmpeg 0.8. This distinction is important for dealing with cases where: different versions of an application are incompatible with each other; the user requires functionality only present in some specific versions; or applications have different functionality for different operating systems.

Whenever a consumer node needs to gather a set of candidate nodes to process gridlets, there are two processes to consider: using sibling or neighbor nodes directly, and using super-nodes.

Each node advertises its own resources and supported applications to the peers comprising its neighborhood. Upon receiving this information, neighbor nodes calculate, with their own judgment, the global rating of the announcer node. This judgment consists of a weighted combination of the measured availability of every single resource. Then, the global rating of the announcer is stored, and when a node needs to execute gridlets, it will prefer providers with higher global rating.

If the neighborhood is not sufficient to process the whole job, then the consumer node requests its primary super-node to find more available nodes. In turn, the primary super-node forwards the request to its counterpart node responsible for the family of the application required to process the job. Upon receiving this request, the super-node looks up on its internal table and tries to gather a set of available nodes that have the job application. This may involve asking other super-nodes with greater aggregated capability for the set of their specific nodes that have the job application. In this way, the consumer node can directly contact provider nodes to send gridlets and save network hops that would otherwise increase the latency of the transmission.

2.7. Gridlet-result storage

To persist gridlet results, this component provides a distributed networked storage system which is maintained by the network nodes. As machines may fail, there is a replication factor allowing equal data to reside in different nodes. The main purpose of this component is to reduce resource utilization, namely CPU and bandwidth, by avoiding redundant execution of gridlets that were already computed in the past.

In practice, this component forms a distributed hash table (DHT), with the typical two operations of lookup and store, mapping gridlet input data digests into gridlet-results/output data. The lookup is performed for every newly created gridlet; and the store is performed every time a gridlet result is deemed trustworthy. Both operations are performed by the consumer node. Furthermore, the maximum lifetime of stored results, as well as the overall size of the storage and replacement policy, can be parameterized.

This storage system reveals itself more useful in small communities (or overlays) where users have quite the same applications installed and work in projects with similar data (e.g. an animation studio or a community of hobbyists working on a same project). For this reason, this storage component can be deactivated, thus reducing the overall system overhead.

3. IMPLEMENTATION

This section details relevant implementation aspects of the architecture previously described. In particular, the used technology, adaptation of applications, EE, and resource management and discovery.

3.1. Used technology and integration

To favor portability, our middleware is developed in Java and runs on top of a Java Virtual Machine, which is available for the most common operating systems and computer architectures. The base platform is intended to be executed in a single process, albeit non-core components can run in isolated processes or different machines. In this case, additional code inside the core components is required to handle the inter-component communication, which can be done, for instance, through RPCs or web services.

With respect to the network, we adopted FreePastry[2] as an implementation of the Pastry P2P overlay. This particular implementation is made in Java and provides several components to create and manage a Pastry overlay network with numerous nodes.

[2] http://www.freepastry.org.

With regard to the GRS component, we resorted to PAST [5], which is a large-scale, P2P archival storage facility. The integration of PAST with our platform was easily done, since PAST is intended to work upon FreePastry. Furthermore, the key to look up for gridlet results is the SHA-1 hash generated from the gridlet description, which includes the input and configuration data.

3.2. Application adaptation engine

Owing to transparency and portability key issues, we do not expect users and developers to adhere to yet another specific API, and to modify applications to interact with the framework and with internal mechanisms (e.g. RD, allocation, bidding). Instead, we rely solely on the ability to understand data formats (via adapters). We modify the input files, splitting them, aware of format intrinsics, creating gridlets that wrap the file chunks/fragments in a way that they appear as correct (yet smaller) files to the applications. We perform similar complementary transformations during the result aggregation process to provide a correct output file to any application (e.g. a video player).

We target mostly CPU-intensive Bag-of-Tasks jobs, where any application carries out processing over input files and generates output files, and these are easily adapted. In this case, regardless of the complex formats, the intrinsic parallelism is easier to extract and leverage, once the format issues are handled. Regarding other categories of jobs (e.g. hierarchical jobs, workflows) we can also integrate the execution of their tasks. In such scenarios, heterogeneity will imply that all the formats involved must have the corresponding adapters for C^3P. Regarding the dependencies between tasks, these are exposed naturally, e.g. as gridlets for higher-level jobs can only generate results once all the results of the lower-level gridlets have become available and aggregated; and downstream tasks in workflows can only be started once the gridlet results of upstream tasks have also been dealt with accordingly.

When splitting an input file to create gridlets, the first step is to parse that file and construct an auxiliary tree in accordance to the XML-based format descriptor specified for the application. This tree represents the structure of the file, where the nodes are logical parts of the file and the leafs represent pointers to the data contained in the file.

Then, this tree is manipulated through sequences of CRUD operations (create, insert, update, delete tree nodes before, after, or between specified elements or tokens) in order to create partitions in several coherent branches and generate smaller trees. These CRUD operations are defined in the format descriptor and must include all the necessary header analysis and reconstruction, patching and structural modifications (e.g. moving blocks across the file). After this transformation, trees are serialized into files and encapsulated inside gridlets as regular files of their respective format.

As for the merging, the process is reverted. Every single output file is parsed and converted to a tree structure according to the defined application grammar. Afterwards, all trees are merged into one through the CRUD operations and the corresponding final output file is built. Moreover, the merging process may (i) start from an empty output file, (ii) start with the result created from the response to the first request or (iii) start with the result created from the response that arrived first.

The entire process for an example ray-tracing application is globally described in Fig. 5. Furthermore, this component was developed in Java and the Saxon XPath engine[3] was used to read and process XML configuration files.

Due to space concerns, we do not provide the full details of this approach in the main body of the paper. In the figures and listings in Appendix, we provide an example of an XML configuration file for an adapter to AVI files which is able to transparently partition a movie file. We also offer the full grammar for specifying applications file formats and the transformations for splitting and aggregation. It involves some knowledge of the formats, that can be derived from specifications or descriptions, but it enables transparent speedups and prevents modifying all the applications using that format.

3.3. EE and supported applications

The EE component can be configured to use sandbox technology, like LinuxSandboxing[4] or sandboxie,[5] in order to create a safe environment, controlling resource usage and protecting the operating system from misused applications. Further, Sandbox is a security mechanism for running programs independently of the rest of the system. It also allows to run multiple instances of a same application on the same OS instance that otherwise would not be possible for some cases. It is often used to execute untested code, or untrusted programs from unverified third parties, suppliers, untrusted users and untrusted web sites since it creates an isolated environment with possibly resource control. For gridlets that are to be processed by Java applications, we can bypass this (if and only if the user has the Java VM installed, and so decides by defining the appropriate security policies to access local resources) and we need not execute the Java VM on top of another virtualization framework. The EE can be configured to use different systems (also LXC[6] in Linux), thereby indicating the command necessary to launch the environment along with all necessary configuration parameters.

This component can take advantage of multi-core CPUs (SMP), as each launched application corresponds to one new process that can be easily scheduled to available cores by

[3] http://saxon.sourceforge.net/.
[4] https://code.google.com/p/chromium/wiki/LinuxSandboxing.
[5] http://www.sandboxie.com.
[6] https://linuxcontainers.org/.

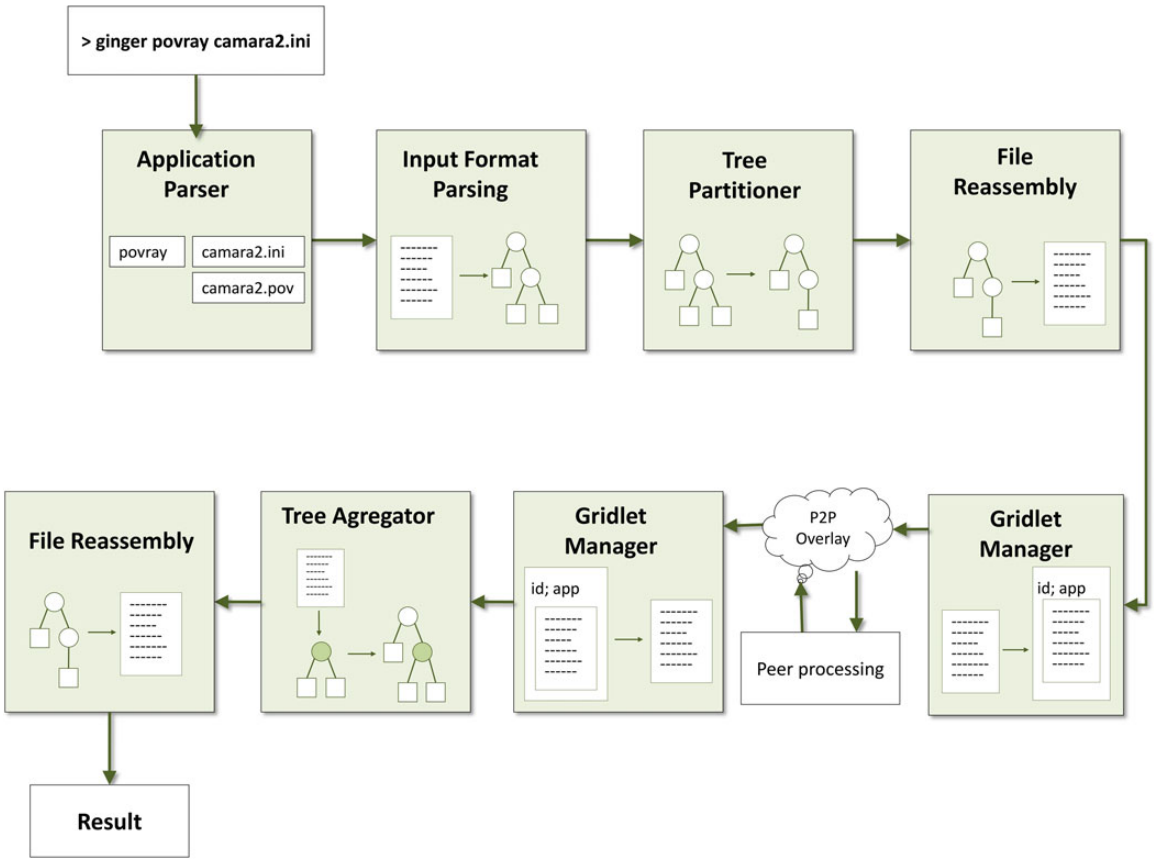

FIGURE 5. C^{3}P application adaptation pipeline.

the operating system. In this way, more than one application may be running at the same time without great performance penalties.

Currently, the kind of applications allowed by this system should either be parameterized through the command line or receive a script or configuration files as input (i.e. parameter sweep or batch file processing). They should also easily generate output files and report errors. Moreover, it should be possible to create independent tasks from those applications.

3.4. Resource management and discovery

To measure the capability and availability levels, every considered resource is assigned with a real number from 0 to 1, where 0 means that the resource is unavailable and 1 that the resource is powerful and has very good availability. Thus, a machine considered as *very good*, and ranking 1 in each resource, would have its resource availability levels at least like <cpu, ram, bandwidth>=<80GFlops,8GB,360Mbps>. These values are not the overall capacity of the machine, but the capability (or availability) at a given time (e.g. a machine with 16 GB of memory, albeit with only 8 GB available). We simply obtain the normalized values by dividing the current resource availability of a machine by our *very good* reference. For example, a machine with resources <cpu, ram, bandwidth>=x GFLOPS,y GB,z Mbps> is normalized to <x/70, y/8, z/360>. If any resource has higher capability than the reference, then the normalized value is automatically 1, since we do not make distinctions above the *very good* reference.

Nodes periodically send update messages to all of their neighbors in order to announce their presence and resources. More precisely, these messages contain the node identifier, its supported applications (i.e. application and family hashes) and its resource availability levels (normalized). In turn, neighbors receive these updates and calculate the global rating of a node.

The global (among all types of resources) rating of a node is obtained through a simple additive model [6]. In this way, nodes define the relative importance of each resource by defining weights (using methods like the swing weights). With them, it is then possible to make a weighted sum and obtain the global availability value (Equation (1)).

$$V(a) = \sum k_j.v_j(a) \quad \text{with} \sum k_j = 1 \quad \text{and } k_j > 0. \quad (1)$$

$j = \{CPU, RAM, bandwidth\}$;
$V(a)$ global value of node a;

$v_j(a)$ partial value of node a in the resource j;
k_j weight of the resource j.

Every time a super-node receives an update message from its neighborhood, it contacts all other super-nodes responsible for the application families therein contained, so that they can update their internal tables with the new aggregated availability of that node for a given application.

3.5. Application programming interfaces

The APIs are organized as follows. Each layer has a unique interface, implemented by its corresponding core component, that is public and visible to other layers. These layer interfaces aggregate (or extend) all individual component interfaces relative to that layer. Components in different layers can only interact through the layer interface and not directly through the individual component interface. Moreover, a component interface should be kept unmodified if the implementation of its component changes.

This organization of interfaces allows the easy reconfiguration and extension of the architecture components; and abstracts in two levels (layer and component interfaces) the specific component (or layer) implementation that can be then changed without compromising the interactions between the other components.

4. EVALUATION

In this section, we present the evaluation of the C^{3}P framework. Components were tested both separately and combined together in simulated and real environments in order to obtain partial and overall performance values and identify possible sources of overhead. All tests were conducted using machines with an Intel Core 2 Quad CPU Q6600 at 2.40 GHz with 7825MB of RAM memory. As for the network, all nodes were within the same LAN and the available bandwidth was around 100 Mbps.

In Section 4.1, we evaluate the impact of adapting applications in the overall performance of the system. We then assess, in Section 4.2, the reliability of the system when malicious nodes collude by returning wrong gridlet results. Section 4.3 explores how the incentive mechanisms, CR, affect the reliability and collaboration of nodes. Following, Section 4.4 evaluates the efficiency of the RD mechanisms; and, finally, the overall performance of the system is assessed when facing real end-to-end application environments (i.e. not simulated), in Section 4.5.

4.1. Transparent adaptation of applications

In this section, we analyze the impact of the AAE component in the overall system performance, thereby identifying overheads and demonstrating how they vary.

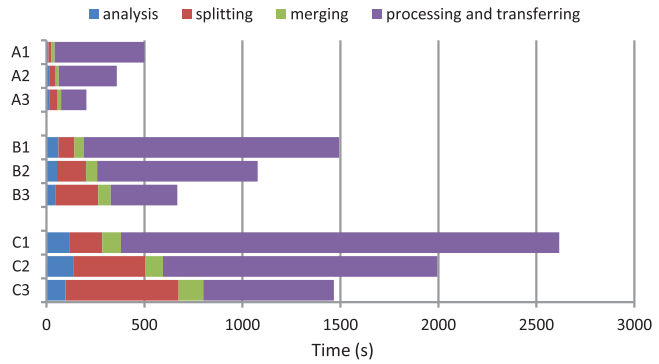

FIGURE 6. Execution time of video compression for one to three nodes with input files with different sizes.

To carry out the tests, we resorted to video transcoding jobs using the ffmpeg application,[7] which, besides being CPU-bound, is relatively I/O intensive and usually deals with large files that either are given as input or produced as output. In particular, for input, we fed the application with video files of type AVI, which represents a complex format with an intricate internal structure. The partition was applied over the key frames, which are good splitting points as they are independent of each other.

For transparency, C^{3}P must address that: (i) key frames are not placed at equidistant offsets but are rather detected within the inner structure of an AVI file; (ii) the ffmpeg application deals with proper AVI files and not with chunks of data between key frames; thus, transparent adaptation is a challenge not without CPU demands.

During evaluation we performed three jobs, a, b and c, consisting of the transcoding of three video files, with the same complexity and sizes 100, 300 and 500 MB, from the mpeg4 xvid codec to the $\times$264. Each job was executed for one to three nodes (designated as a1, a2, a3, b1, b2, b3, c1, c2 and c3).

Results are shown in Figs. 6 and 7, and there are two important considerations to make: (i) the overall processing and response times are conservative, i.e. taken from the gridlet that takes the longest time to produce its result (streaming could have started with only a fraction of the gridlets executed); and (ii) this component is still not fully optimized with regard to parsing and interpreting xml-based heavy grammars.

The first observation we can take is that the time taken for adapting applications increases with the size of the input (a1 with 8%, b1 with 13%, c1 with 14%), and also increases for higher number of nodes, especially the splitting process time. This process of splitting is the most intensive and time consuming due mainly to the generation and management of new trees (representing the internal structure of the videos), which can be significantly enhanced in future versions.

[7] http://www.ffmpeg.org.

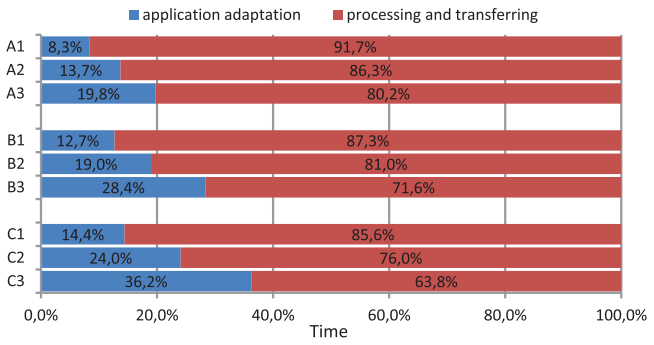

FIGURE 7. Comparison between application adaptation and processing time for one to three nodes with input files with different sizes.

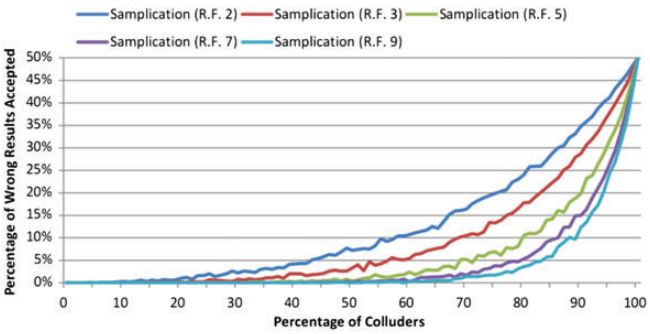

FIGURE 8. Samplication: percentage of wrong results accepted for several replicated factors in a scenario where returned results are 50% corrupted.

Besides lacking optimization, every time a new splitting is required, the source tree is copied again in order to apply the transformations over a clean version of this tree. Since 100 MB videos produce a tree of around 50 000 nodes, it is a slow process to make a copy, apply the transformations and produce the gridlet. In fact, we can say with some confidence that this is the main problem and not the splitting transformations, since this is much less noticeable in the merging process that only works with one tree. Nonetheless, this process is fully driven by declarative XML format specifications that are run in Java code and so, while much optimization is still possible, the flexibility and transparency are demonstrated.

That said, we can expect in future versions a similar evolution of the time taken for the splitting and merging processes. Nonetheless, the current impact this component causes is quite small, around 20% of the overall time taken, and, as the number of nodes increases, the gain in parallelism is much higher than the loss in splitting gridlets. Moreover, c3 obtained 36% on application adaptation due to the processing time that was substantially decreased. Thus, we must stress two things: (i) adaptation weight increases relatively when execution speedups increase, and this is only natural. It shows that there is obviously a ceiling to the speedups an application can achieve (regardless of how many and small gridlets are created) when the costs of adaptation start to dominate, and (ii) as detailed next, the relevant metric for users will be the relative speedup of the entire application execution and not the specific breakdown of where time is being spent. Furthermore, XML parsing and application adaptation can be significantly optimized if they are also catered to be optimized for parallel execution, that we are further developing.

Furthermore, the experimental evaluation demonstrated that popular commodity applications, such as ffmpeg, can be efficiently and transparently adapted to execute several tasks in parallel over a distributed environment. Results show that this approach is feasible, as it allows users to execute jobs remotely, in parallel fashion, without the need to modify applications' source or binary code. Performance results identified the (not great) overheads related with splitting and aggregation of complex file formats (i.e. MPEG4), which are not present in other applications based on parameter sweep (e.g. ray-tracing animations). Naturally, speedups are more favorable in jobs with higher CPU/IO ratio.

4.2. Gridlet-result verification

This section describes the evaluation of the GRV component, which makes use of a technique that combines replication and local sampling (samplication). As previously explained, unequal fragments of replicated gridlet results are tested against local execution of those same fragments. If results are equal, the fragments are randomly picked. In the end, only results matching the local sample are kept.

As other typical techniques, samplication can be fooled if a group of nodes decides to collude. This may happen if the replicated results of a same gridlet are equal to one another and have some fragments that are corrupted. In this way, the samples to be locally executed are randomly picked and, therefore, the (wrong) results can be accepted if the samples match non-corrupted fragments. As such, we simulated a scenario to assess the percentage of wrong results accepted, in the presence of colluding nodes, for several replication factors. Additionally, each returned result was corrupted in random 50% of its content regarded as a bitstream.

As depicted in Fig. 8, samplication is effective in maintaining the wrong results accepted very low for small, and even medium (30–70%), groups of colluding nodes. Note that in a real environment colluding nodes are typically a minority. When colluding nodes correspond to $<50\%$, the wrong results accepted are below 10% (and below 5% for replication factors >2), which is very good when compared with other techniques. For instance, other tests we conducted, for standard replication without sampling, show that the wrong results accepted are $>10\%$ when colluding nodes correspond to 20% or more.

Furthermore, results have also shown in this simulation that the incurred overhead of locally running samples, and identifying and aggregating correct results, assumes a good

compromise with the reliability of the gridlet results. In addition, replication is considerably improved by the addition of the sampling model, as voting quorum schemes are completely vulnerable to colluding nodes. Thus, on average, samplication will incur lower resource usage, to ensure a given level of trust, when compared with simply replicating gridlets or entire tasks in order to achieve quorums and determine the trusted result.

4.3. Currency and reputation

In this section, we present the average results of our two most relevant simulations regarding the CR component. The first simulation, Zero Incentive Mechanisms, consists of a scenario to assess the impact and leverage that malicious nodes have on a system with no incentives. In this scenario, 10% of the nodes act maliciously by illegitimately trying to maximize their gain. These nodes reject all requests for resources, collude to improve their reputation and utilize the initially awarded currency to obtain resources from other nodes, simply creating new identities when needed.

On the other hand, the second scenario, Full Incentive Mechanisms, intends to measure the effectiveness of our incentive mechanisms, comparing the results achieved by the malicious nodes with the ones from the first scenario. Besides the general settings, these incentive mechanisms also encompass the following activities: (i) progressively award CR, where nodes start with a low initial budget and periodically receive more currency (instead of starting with all of it), which forces users to accumulate currency and share more resources; (ii) periodic decrease of reputation, to force nodes to share resources and (iii) collusion detection, where super-nodes maintain a record of transactions of their child nodes to verify possible collusion between nodes.

The simulation settings are described in Table 1. We used PeerSim with the different types of nodes uniformly randomly distributed over the overlay and the results are averages of 10 simulations.

The average results of the performed simulations are presented in Table 2. In addition, we obtained a similar elapsed time in both simulations, meaning that the application throughput is not affected by the incentive mechanisms.

As can be seen, the success of attacks from faulty nodes is greatly reduced with the incentive mechanisms, reducing the completed exchanges of malicious users, from twice as much as those accomplished by regular nodes to just 11%. The results are very positive and show that faulty nodes are identified and do not have any advantage or leverage over legitimate nodes.

Furthermore, the small initial budget and progressive introduction of currency results in a slow start even for honest users. For that reason, the number of successful transactions may appear to be weak but, as the number of interactions increases and time advances, that initial impact is softened.

TABLE 1. Simulation parameters.

Parameter	Zero incentive mechanisms	Full incentive mechanisms
Initial budget	100	1
Initial reputation	50	50
Maximum cost of a resource	5	5
Probability of a node being faulty	10%	10%
Probability of a faulty node making fake requests or acceptances	10%	10%
Probability of a faulty node badmouthing or praising	10%	10%
Probability of collusion between faulty nodes	25%	25%

TABLE 2. Simulation results.

	Zero incentive mechanisms	Full incentive mechanisms
Number of nodes	2249	2036
Expelled nodes	0	308
Super nodes	0	5
Faulty nodes	211	27
Exchanges attempted	21 960	19 294
Exchanges failed	0	850
Reputation (correct nodes)	72	60
Budget (correct nodes)	100	28
Successful exchanges (correct nodes)	8.27	7.98
Reputation (faulty nodes)	74	36
Budget (faulty nodes)	109	3
Successful exchanges (faulty nodes)	17.14	0.9

Through Fig. 9 it is possible to observe that routing gridlets while taking into account reputation information improves the efficiency, i.e. the average number of hops taken per time unit to process jobs. The gain was less accentuated in the first iteration because the consumer node had not acquired by then the reputation information regarding its neighbors. Starting on the second iteration, there is a strong reduction in the number of re-transmissions performed (since we are able to better avoid malicious nodes), and we can see that the convergence for the optimal point is fast.

4.4. Resource discovery

This section presents the evaluation of the RD mechanisms implemented in C^3P for a Pastry overlay network. As these mechanisms take place soon after the start of a job, they can incur overhead to the overall application performance, especially due to the messages that are exchanged with super-nodes to locate resources in farther domains. As such, we

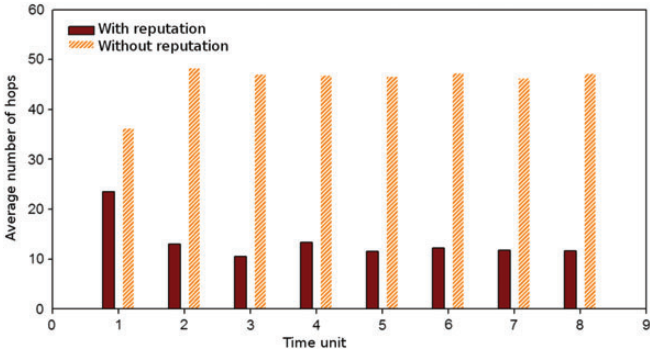

FIGURE 9. Average number of hops per time unit with and without using reputation.

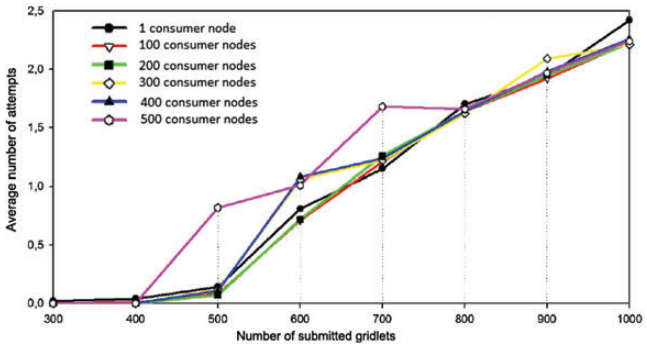

FIGURE 11. Number of attempts to locate available resources.

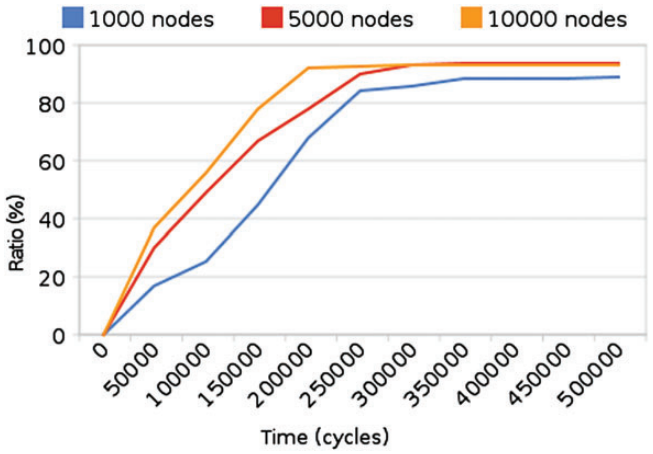

FIGURE 10. RD effectiveness.

designed scenarios to assess the efficiency (in terms of hops and time) and number of messages exchanged.

All tests were executed in a simulated environment and the time units refer to the number of simulated cycles processed. To get representative results, we performed 500 000 cycles, during which nodes can send or receive messages with a low probability. Also, we assumed a uniform distribution of applications amongst all nodes.

The first test is intended to show the effectiveness of the RD component. More specifically, the percentage of executed gridlets (or ratio) during the simulation time (i.e. resources that are located, perform work and later return results). As we can see through Fig. 10, a large number of gridlets do not find any appropriate node initially. This happens because nodes are still joining the network and, as time goes by, the amount of gridlets that successfully locate the required resources increases, stabilizing more rapidly for larger number of nodes (as expected). Moreover, all curves converged to 100, meaning that sent gridlets will eventually find available resources with the required applications to get executed.

In another scenario, we assess the average number of attempts consumer nodes make in order to locate available resources and get their gridlets effectively processed. When a request for resources returns a set of candidate available nodes that do not have an overall capacity sufficient to attend all gridlets, an attempt is made (albeit some of the gridlets were already dispatched). This simulation was performed with 500 nodes, each with capacity to process only 1 gridlet at a time.

Through Fig. 11 we may see that, as the number of submitted gridlets' increases from 600 to 1000, the number of requested resources is greater than the number of resources that were provided within the network; therefore, more attempts were made until the end of the first gridlets' computation. Below 500, the number of attempts is almost 0 (gridlets were served immediately in the first request), except for the case of 500 consumers. This divergence happened due to the concurrency of 500 nodes trying to compete for the same resources, albeit this impact was minimal (only 0.7 more attempts on average).

In Fig. 12, we have the average hops a gridlet travels until finding an available and capable node to execute it, obtained for a set of simulations. Through it, we can see that the number of hops evolves logarithmically with the number of nodes, which demonstrates scalability. There is a number of hops spent in super-peer redirection, but it levels rapidly as the number of super-nodes stabilizes and they become known amongst themselves as to become nearly insignificant.

Figure 13 shows the number of messages exchanged across the network by their type. The amount of messages spent in querying super-nodes and the subsequent replies is relatively high (as they account for almost 10% of all network traffic), especially due to the degree of churn introduced in the simulation.

With regard to the advertising messages that nodes send periodically to announce their resources, they increase quadratically with the number of nodes: n nodes will exchange $2n(n-1)$ messages. Note that this information could also be piggy-backed on any other type of message nodes exchange, namely when relaying gridlet requests or results among nodes.

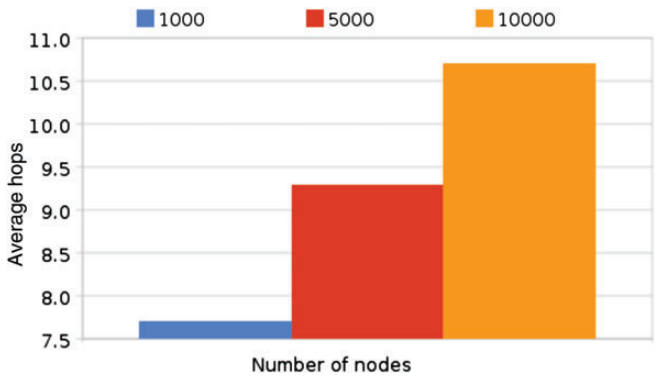

FIGURE 12. Average number of hops.

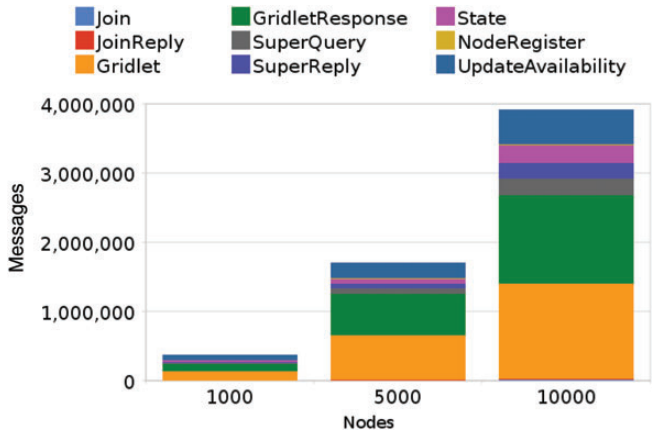

FIGURE 13. Number of exchanged messages.

4.5. Overall performance in real environment

In this section, we evaluate the C^{3}P framework as a whole and assess the effective gains in terms of performance in a real environment. More precisely, we measure and analyze the speedup[8] of applications for a set of key scenarios. To execute the tests, we relied on two applications and three jobs, listed as follows.

(1) A POVray image to be rendered in which each gridlet computes a certain number of line chunks with different complexity. Owing to that carried complexity, some gridlets can be computed faster than others.
(2) A POVray image to be rendered, albeit the computational cost of each task is the same and gridlets should be completed at the same time. Additionally, this job is less computationally heavy than the previous one, i.e. with the highest number of nodes the gains of the parallelization may be lower.

[8] $S_p = T_1/T_p$, where S_p is the speedup with p nodes, T_1 is the execution time with one node and T_p the execution time with p nodes.

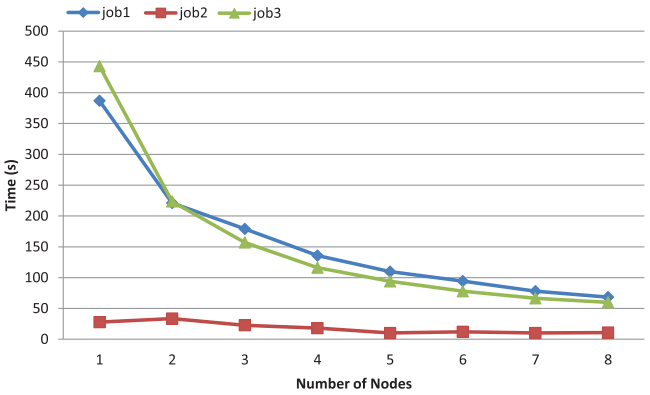

FIGURE 14. Time elapsed for job 1, 2 and 3 using from one to eight nodes.

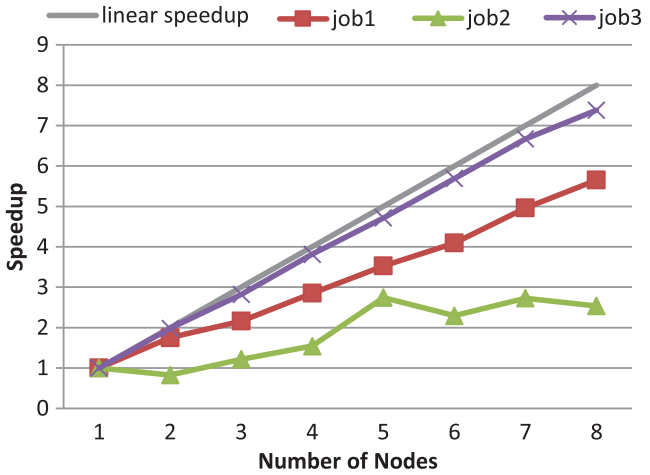

FIGURE 15. Speedup obtained for job 1, 2 and 3 using from one to eight nodes.

(3) A Monte Carlo simulation for a sum of several given uniform variables in which the outcome is an image containing a linear chart. Each task generates a lot of random numbers and groups them into classes. Then, the classes are summed between tasks, and a Monte Carlo curve is drawn. Additionally, all gridlets should take the same time to be computed.

Among all executed jobs, these ones were carefully chosen for this article as they highlight the three different, and most important, impact levels that jobs can have on performance. For each of these jobs we made eight trials. In the first trial, we used one node (i.e. no parallelism, as the consumer node is also the only provider), in the second two nodes and so on until we had eight nodes in total.

From Figs. 14 and 15, we may see that job 1 and job 3 performed better. In particular, the speedup of job 3 was almost linear. This happened due to two major reasons. First, each generated gridlet for job 3 had the same computational

complexity, unlike job 1, and therefore the work was evenly distributed among provider nodes. In this way, all gridlets were completed at the same time and the allocated resources were fully utilized. In job 1, machine providers were loaded differently, and thus some nodes became free while others were still computing gridlets.

Second, the input and output of job 1 consisted of files whose size was in the order of MB, whereas the application of job 3 only requires a configuration file as input and generates small-sized text files as output; i.e. the transmission of data throughout the network had a significant impact on the performance. This means that jobs with higher CPU/bandwidth ratio will naturally perform better.

Nevertheless, our results have also shown that the small-sized gridlets and the pipelining approach (described in Section 2.3) can improve performance. Considering only one worker machine as reference, one gridlet with 500 MB took about 30 s to be sent, 1200 s to be processed and 22 s to be returned with an output of 250 MB. When we split that gridlet into smaller ones of 1 MB, we obtained 1200 s of processing time (with only one worker machine as before), $<$15 s of transmission time (this time was taken from the first sent gridlet and the last returned gridlets) and $\sim$3 s of additional overhead. Albeit the gain here was only 3% (with 100 Mbps of bandwidth), greater gains are expected for wide-area networks (like the Internet).

Considering the worst result, job 2 is an example of a job that probably is not worth to parallelize, since its computational complexity is not high enough to compensate the incurred framework overhead.

In the next section, we also offer some comparison with other relevant systems following their description and brief analysis.

5. RELATED WORK

In this section, we review relevant solutions, within the current state of the art, regarding the main topics approached in this work: transparent adaptation of applications, result verification, CR and RD. We then contrast the most successful and complete systems with our solution.

5.1. Constituent topics and systems

Transparent adaptation of applications for distributed execution: To ease the interactions with grid schedulers, earlier work on Nimrod [7], XtremWeb [8] and Ganga [9] shows specialized developments on user interfaces for creation of independent tasks. These tools allow the easy creation of parameter sweep jobs, where each task processes part of an input domain. The partition of this domain is done by assigning different input files to different tasks, or the same input file to all tasks, but dividing some parameter intervals (e.g. chunks of data to process in each task). Unlike our approach, the partition of large data files into smaller data units is not possible, which makes these tools somehow limited.

The work in [10] proposes an interesting generic XML-based tool for data partitioning in work units that are scheduled in nodes across the Grid. Resource estimation is based on probing remote nodes with sample data which assumes full trust. Therefore, it does not address the same P2P environments our work does. Their XML descriptions for data partitioning allow for alternatives of fixed-sized units, cutting on known separators, user-provided files with cut-points or delegating to external tools. Result-gathering is left to the user. Our XML format descriptions are more flexible handling other aspects of current formats, such as maintaining and adapting headers, and identifying parts of files that must be in the same unit (full and predicted frames in MPEG).

The majority of distributed cycle-sharing systems rely on specific and application-dependent mechanisms for splitting and merging data that usually will not fit in new applications that users would like to include in the system. For example, BOINC requires application owners to create the necessary code to partition the input domain of the problem in small work units, each describing the computation to be performed on every single remote node.

In the same way, Ewert [11] suggests applying grid technologies for analyzing multimedia content and video files, such as detecting transitions and faces; however, the proposed prototype performs the splitting of video scenes through mechanisms that are hard-coded in the middleware, thus not allowing the easy addition of new data formats or the use of the infrastructure to solve different problems.

In our approach, users have not to develop any code to adapt applications into the system; instead, they have only to provide a grammar in a high-level language that defines the rules of the partition and aggregation of tasks and data. In addition, this grammar could be downloaded automatically from a repository for a given application, precluding any user intervention in this process.

Result verification: One of the most typical and effective methods to identify wrong results is through redundant execution, on multiple nodes, of a same task and comparison of its results based on a voting quorum scheme. It is almost impossible for a fault or byzantine behavior in different machines to produce the same wrong result; so, in these cases, the wrong results are easily identified and discarded. Nevertheless, if a group of nodes colludes, this technique may fail to identify wrong results. Also, another drawback of redundant execution is the overhead and resource consumption it generates, since every job is executed at least three times. Most of the systems rely on this standard replication mechanism, such as BOINC.

In [12], is proposed a technique with hash-trees in which provider nodes are forced to calculate a binary hash-tree from their results. The leafs of this hash-tree are partitioned

sequential results. The hash is calculated using two consecutive parts of the result concatenated, starting by the leafs and ending in the root of the tree. Once the tree is complete, the consumer has only to execute a random part of the result (one leaf) and calculate its hash. Then this result is compared against the returned result and the hashes of the whole tree are checked. This method is very effective, as finding the correct hash-tree requires more computation than actually performing the required computation. Although, it does not dissuade providers that are willing to forge their results at any cost. Also, results have to be decomposed (large overhead).

CCOF [13] proposes quiz mechanisms consisting of assigning workunits (like jobs) whose outcome is known beforehand. It describes two types of quizzes, stand-alone and embedded quizzes. Stand-alone quizzes are disguised as regular jobs to check if a node produces the expected result; while the embedded ones are smaller quizzes that are placed hidden into a job. The use of the same quiz more than once can enable malicious nodes to identify them and fool the reputation mechanisms. In addition, the implementation of embedded quizzes tends to be complex in most cases.

CR: With respect to reputation systems, the web site eBay [14] is probably the most famous example. When you buy a product from another user, you are asked to rate the seller according to the quality of the service. If a user receives good reviews, it adds to his reputation and he becomes a renowned seller, attracting more potential buyers (i.e. less risk is involved). The reputation values are stored in a central server considered trusty. Although, there are decentralized alternatives (especially for p2p) such as the Eigen Trust Algorithm [15].

In [16], authors develop and test a framework for propagating trust and distrust, changing the focus from the number of voters to relationships developed over time, much like in the real world, where you trust the opinion of a long acquaintance more than that of 10 strangers. However, this framework may impact significantly the performance of the whole system, as many nodes need to keep the reputation information and propagate it.

The work in [17] presents a mechanism for lightweight and distributed trust propagation, with the intent of implementation in low-capacity devices, such as mobile phones. However, it does not address the particular aspects of resource management in a P2P cycle-sharing system.

Currency used in [18] is considered appropriate for systems without a controlling third party, such as the P2P system, because it has far fewer security concerns than real money. Other options exist, such as bargaining or auctions [19], where the consumers not only have the choice of buying or not, but also influence the value asked. It is up to the resource owner to decide which method is used.

RD and scheduling: Iamnitchi *et al.* [20] have compared different searching methods. It was concluded that a learning-based strategy achieves more performance. Such strategy consists of forwarding a request to the node that answered similar requests previously (i.e. using a possibly large cache). Moreover, results have shown that searching mechanisms that keep a history of past events are more efficient than the ones that do not store any information about other nodes, such as the random walk.

CCOF [21] has tried several approaches, and the one obtaining best global performance was based on a partially centralized P2P overlay. Within this approach, some nodes may acquire a special role in the network and provide a service of lookup for nodes nearby. This way, nodes advertise their profiles and address requests to those super-nodes. Whenever a request is made, super-nodes attempt to match the query with cached profiles and return a set of candidate nodes. Nevertheless, the dynamic placement of these super-nodes is still an open problem.

Cheema *et al.* [22] proposed a solution for exploiting the single keyword DHT lookup for CPU cycle-sharing systems. This solution consists in encoding resource identifiers based on static and dynamic resource descriptions. The static ones could be, for instance, the OS configuration, RAM or CPU speed. While the dynamic descriptions are related to the availability levels of resources, such as the percentage of idle CPU. With this encoding mechanism, it is possible to create a mapping between resources and node identifiers in structured P2P networks, like Pastry, and take advantage of the efficient routing of queries.

In cluster scenarios, RD, allocation and scheduling is a very relevant aspect. Tycoon [23] proposes a distributed market-inspired resource allocation system. In Tycoon, users manage accounts and pay for the resources used, based on the outcome of the matching of bids by requesting users, and offers by resource donors that make advertisements. It has low overhead, leverages virtualization technology to execute tasks, and takes special care regarding network and messaging overhead.

Mesos [24] is a platform dedicated to cluster sharing among different computational platforms, as it is becoming common nowadays. It is able to fulfill high-level policies regarding resource utilization by each competing framework. It handles availability and fault-tolerance concerns mostly automatically, and allows better resource utilization regarding more fine-grained slots of available resources, regarding both time and space.

YARN [25] is a similar more recent system, fostered by Apache for next-generation Hadoop. It has significant performance and resource utilization improvements, and handles several frameworks simultaneously. It also manages availability, fault-tolerance, auditability and caters for locality of data when the jobs and tasks are being scheduled for increased performance. Both Mesos and YARN assume full compliance with a new API and do not resort to virtualization technology which may hinder adoption of similar efforts in non-dedicated desktop machines of users.

Merlin [26] goes a step forward regarding the current scale-up in clusters and proposes specific optimizations for consolidating workloads in many-core machines, namely NUMA machines, specifically addressing multi-core and caching hierarchy issues for resource reconfiguration.

Comparison with C^{3}P: Evaluating the same workloads with very different platforms is an engineering feat in itself, more difficult even when the code of some systems is not available or has completely different requirements and assumptions. We thus focus on the computational and message complexity of the approaches, and the overhead introduced to working nodes.

Naturally, regarding full-fledged time-trail assessment, they are not comparable. C^{3}P is a P2P system, whereas Mesos, YARN and Tycoon rely mostly on a master–slave model meant to be run in resourceful datacenters, with dedicated resources and very low latencies. Therefore, regarding total execution times, they will be necessarily slower in C^{3}P, mainly due to the cost of data transfer on wide-area networks (order of magnitude lower bandwidth) and diminishingly due to the cost of adaptation, GC and result aggregation. If run on top of a cluster (not designed for such), C^{3}P will also naturally benefit from this. They also incur lower overhead because they execute jobs as native processes or over light virtualization frameworks, with no overhead to prepare data for processing. C^{3}P incurs in the mentioned data adaptation overhead and will almost always execute over heavier virtualization for increased isolation and user protection.

Tycoon, Mesos and YARN discovery and allocation protocols are based on direct interaction between bidders/requesters and donors, which is acceptable in clusters but difficult to scale to P2P networks. Such requirements or obligations of contacting individual nodes (requesters and/or providers) directly to bid or offer resources, clearly does not scale to P2P networks. For this, we rely on super-peers to drive the network traffic down, and to allow smart concise resource representation for resource and application discovery (another problem not relevant in homogeneous clusters, even if running multiple platforms at once).

Tycoon, Mesos and YARN require the adoption of applications of framework schedulers and RMs to specific APIs. This is somewhat natural in a cluster scenario where developers and system administrators agree to common denominators. It is clearly not the case in large-scale scenarios, such as P2P cycle-sharing systems and community clouds, where most users will only use the same unmodified applications they are already familiar with, and are unable to modify them, and unwilling to execute others they do not know well.

Therefore, for communities, small groups of scientists and even SMEs, who do not possess either the necessary infrastructure to deploy and run resource-intensive dataflow applications or sufficient available monetary funds to contract public cloud computing services, volunteer computing can turn out to be a paramount option. We want to emphasize this clear separation between grid p2p and the typical centralized cluster model.

Therefore, while some of their principles and lessons can be leveraged to other scenarios, by themselves, they cannot be applied to current cycle-sharing systems such as community clouds, mainly because they lack the required transparency and scalability to large-scale networks. Merlin does not consider distributed settings.

C^{3}P has a decentralized (no central node) RD for scalability to large-scale networks, but incorporating hierarchical elements (super-peers) for message efficiency and reduced latency.

5.2. Cycle-sharing systems

Institutional grids: Globus [27] is an enabling technology for grid deployment. It provides mechanisms for communication, authentication, network information and data access, amongst others. The authentication and authorization models are directed to institutions, making it difficult for ordinary users to deploy applications on top of the Grid. In contrast, C^{3}P envisions an open access environment whereby the complexity of getting credentials to use the Grid is reduced.

Condor [28] allows the integration and use of remote workstations. It maximizes the utilization of workstations and expands the resources available to users, by functioning well in an environment of distributed ownership. Condor's jobs rely on executable binary code in which compatible machines are needed in order to run them. Contrastingly, machine heterogeneity in C^{3}P is not a problem, since jobs consist of data files that can be easily read by any computer (i.e. the kind of architecture and operating system are not relevant). In addition, many Condor features require some degree of expertise (i.e. advanced configurations are needed), whereas our platform keeps the vision of simplicity and tries to do all the necessary work with almost no interference from users.

Master-Slave model: BOINC [29] is a platform for volunteer-distributed cycle sharing based on the client–server model. It relies on an asymmetric relationship where users, acting as clients, may donate their idle CPU cycles to a server, but cannot use spare cycles from other clients for themselves. Besides that, setting up the required infrastructure, developing applications and gathering enough cycles could be difficult for an ordinary user. On one hand, users need to have the required skills to create BOINC projects, and, on the other, hand projects should have a large visibility to attract users to participate in. In comparison, C^{3}P is more flexible as users have the same power to both provide and consume idle cycles to and from other machines. Moreover, it is possible to use common and widely used applications with our system.

nuBOINC [30] is a project that attempts to overcome the drawbacks presented by BOINC. It allows one to use idle cycles from other users, through servers and making use of commodity applications. However, C^{3}P relies on a more scalable model that does not require the intermediation of servers. Also, when work units are being distributed,

our platform takes into account the idleness levels of user machines, which is disregarded by nuBOINC.

P2P: CCOF [13] is an open P2P system seeking to harvest idle CPU cycles from its connected users. It shares our goals of reaching the average user by not requiring any kind of membership or negotiations in any organization (i.e. in contrast with institutional grids). Despite that, CCOF is absent in what concerns the adaptation of applications to their system, including not presenting any use case with applications and related performance evaluation.

OurGrid [31] is a P2P network of sites that tries to facilitate the inter-domain access to resources in an equitable manner. Each of these sites comprises grid clusters possibly belonging to different domains. The sharing of resources is made in a way that makes those who contribute more to get more when they are in need. Nonetheless, applications need to be modified in order to run on top of this platform, and the data-based parallelism is not exploited as in C^{3}P. Besides that, machines are not distinguished by their idleness levels, whereas our platform attempts to always select the best available nodes for a job.

6. CONCLUSION

This article addresses a current reality characterized by: (i) underused and powerful computational resources connected throughout possible large and high throughput networks, (ii) general society and non-expert user's computational needs and (iii) computing models oriented for high performance.

We present the design, development and evaluation of the C^{3}P framework, a flexible solution based on the gridlet concept, introducing a new application model that can bridge the gaps between a number of existing infrastructures (e.g. grids, distributed cycle-sharing and decentralized P2P file-sharing), bringing Grid technology to home users. We enable parallel execution of commodity applications in a transparent manner, i.e. without needing to introduce modifications in applications' source or binary code.

A representative selection of relevant solutions are reviewed and it is concluded that none of them entirely covers the objectives of C^{3}P. In particular, regarding the distributed cycle-sharing systems, they fail to reach the common user due to institutional barriers, they are not portable, they require modifications to applications and idleness levels of nodes are disregarded, among other reasons.

In summary, although there is relevant related work and successful projects in the areas of grid computing, distributed cycle-sharing and P2P computing, to the best of our knowledge, none of them provides an application model and middleware framework that offers improved performance, with transparency, to existing applications executed by Internet home users. Thus, we find that C^{3}P, due to its unique characteristics, is a compelling effort, within the current state of the art, to unleash and combine the computational power scattered across different network and architectural settings, and make it available to Internet users.

ACKNOWLEDGEMENTS

The authors wish to thank students Filipe Paredes, João Morais, Pedro Rodrigues, João Paulino, João Neves and Pedro Oliveira for their work and enthusiasm, and Rodrigo Rodrigues, Carlos Ribeiro and João Nuno Silva for their teamwork during the various stages of this work.

FUNDING

This work was supported by national funds through Fundação para a Ciência e a Tecnologia with reference UID/CEC/50021/2013.

REFERENCES

[1] Rowstron, A. and Druschel, P. (2001) *Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems*, Lecture Notes in Computer Science 2218. Springer, Berlin, 329–350.

[2] Stoica, I., Morris, R., Karger, D., Kaashoek, M.F. and Balakrishnan, H. (2001) Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. *SIGCOMM '01: Proc. 2001 Conf. Applications, Technologies, Architectures, and Protocols for Computer Communications*, New York, NY, USA, pp. 149–160. ACM.

[3] Ratnasamy, S., Francis, P., Handley, M., Karp, R. and Schenker, S. (2001) A Scalable Content-Addressable Network. *SIGCOMM '01: Proc. 2001 Conf. Applications, Technologies, Architectures, and Protocols for Computer Communications*, New York, NY, USA, pp. 161–172. ACM.

[4] Dongarra, J.J., Luszczek, P. and Petitet, A. (2003) The Linpack benchmark: past, present, and future. *Concurrency Comput. Pract. Exp.*, **15**, 803–820.

[5] Rowstron, A. and Druschel, P. (2001) Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. *SIGOPS Oper. Syst. Rev.*, **35**, 188–201.

[6] Goodwin, P. and Wright, G. (2004) *Decision Analysis for Management Judgment* (3rd edn). John Wiley & Sons, Ltd.

[7] Abramson, D., Sosic, R., Giddy, J. and Hall, B. (1995) Nimrod: A Tool for Performing Parameterised Simulations using Distributed Workstations. *Proc. 4th Int. Symp. High Performance Distributed Computing (HPDC '95)*, Washington, DC, USA, pp. 112–121.

[8] Germain, C., Nori, V., Fedak, G. and Cappello, F. (2000) Xtremweb: Building an Experimental Platform for Global Computing. *GRID '00: Proc. 1st IEEE/ACM Int. Workshop on Grid Computing*, London, UK, pp. 91–101. Springer.

[9] Egede, U., Moscicki, J., Patrick, G., Soroko, A. and Tan , C. (2005) Ganga User Interface for Job Definition and Management. *Proc. 4th Int. Workshop on Frontier Science: New Frontiers in Subnuclear Physics*, Italy, September. Laboratori Nazionali di Frascati.

[10] van der Raadt, K., Yang, Y. and Casanova, H. (2005) Practical Divisible Load Scheduling on Grid Platforms with APST-DV. *Proc. 19th IEEE Int. Parallel and Distributed Processing Symp. (IPDPS'05), IPDPS '05*, Washington, DC, USA, IPDPS '05, pp. 29b–29b. IEEE Computer Society.

[11] Ewerth, R., Friese, T., Grube, M. and Freisleben, B. (2004) Grid Services for Distributed Video Cut Detection. *ISMSE '04: Proc. IEEE 6th Int. Symp. Multimedia Software Engineering*, Washington, DC, USA, pp. 164–168. IEEE Computer Society.

[12] Du, W., Jia, J., Mangal, M. and Murugesan, M. (2004) Uncheatable Grid Computing. *Proc. 24th Int. Conf. Distributed Computing Systems (ICDCS'04), ICDCS '04*, Washington, DC, USA, pp. 4–11. IEEE Computer Society.

[13] Lo, V., Zappala, D., Zhou, D., Liu, Y. and Zhao, S. (2004) Cluster Computing on the Fly: P2p Scheduling of Idle Cycles in the Internet. *Proc. 3rd Int. Conf. Peer-to-Peer Systems, IPTPS'04*, Berlin, Heidelberg, pp. 227–236. Springer.

[14] Dellarocas, C. (2001) Analyzing the Economic Efficiency of eBay-like Online Reputation Reporting Mechanisms. *Proc. 3rd ACM Conf. Electronic Commerce, EC '01*, New York, NY, USA, pp. 171–179. ACM.

[15] Kamvar, S.D., Schlosser, M.T. and Garcia-Molina, H. (2003) The Eigentrust Algorithm for Reputation Management in p2p Networks. *Proc. 12th Int. Conf. World Wide Web, WWW '03*, New York, NY, USA, pp. 640–651. ACM.

[16] Guha, R., Kumar, R., Raghavan, P. and Tomkins, A. (2004) Propagation of Trust and Distrust. *Proc. 13th Int. Conf. World Wide Web, WWW '04*, New York, NY, USA, pp. 403–412. ACM.

[17] Jesi, G.P. (2007) Secure gossiping techniques and components. PhD Thesis, Technical Report UBLCS-2007-08. University of Bologna (Italy), Department of Computer Science.

[18] Vishnumurthy, V., Chandrakumar, S., Ch, S. and Sirer, E.G. (2003) Karma: A Secure Economic Framework for Peer-to-Peer Resource Sharing. *Proceedings of the 1st Workshop on Economics of Peer-to-Peer Systems*, Berkeley, CA, USA.

[19] Rolli, D., Conrad, M., Neumann, D. and Sorge, C. (2005) An Asynchronous and Secure Ascending Peer-to-Peer Auction. *Proc. 2005 ACM SIGCOMM Workshop on Economics of Peer-to-Peer Systems, P2PECON '05*, New York, NY, USA, pp. 105–110. ACM.

[20] Iamnitchi, A. and Foster, I. (2004) A Peer-to-Peer Approach to Resource Location in Grid Environments. *Grid Resource Management: State of the Art and Future Trends*, Norwell, MA, USA, pp. 413–429. Kluwer Academic Publishers.

[21] Zhou, D. and Lo, V. (2004) Cluster Computing on the Fly: Resource Discovery in a Cycle Sharing Peer-to-Peer System. *CCGRID '04: Proc. 2004 IEEE Int. Symp. Cluster Computing and the Grid*, Washington, DC, USA, pp. 66–73. IEEE Computer Society.

[22] Cheema, A.S., Muhammad, M. and Gupta, I. (2005) Peer-to-Peer Discovery of Computational Resources for Grid Applications. *GRID '05: Proc. 6th IEEE/ACM Int. Workshop on Grid Computing*, Washington, DC, USA, pp. 179–185. IEEE Computer Society.

[23] Lai, K., Rasmusson, L., Adar, E., Sorkin, S., Zhang, L. and Huberman, B.A. (2005) Tycoon: an implementation of a distributed, market-based resource allocation system. *Multiagent Grid Syst.*, **1**, 169–182.

[24] Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A. D., Katz, R. H., Shenker, S. and Stoica, I. (2011) Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In Andersen, D.G. and Ratnasamy, S. (eds), *Proc. 8th USENIX Symp. Networked Systems Design and Implementation, NSDI 2011*, Boston, MA, USA, March 30–April 1. USENIX Association.

[25] Vavilapalli, V.K. *et al*. (2013) Apache Hadoop YARN: Yet Another Resource Negotiator. In Lohman, G.M. (ed.), *ACM Symp. Cloud Computing, SOCC '13*, Santa Clara, CA, USA, October 1–3, p. 5. ACM.

[26] Tembey, P., Gavrilovska, A. and Schwan, K. (2014) Merlin: Application- and Platform-Aware Resource Allocation in Consolidated Server Systems. *Proc. ACM Symp. Cloud Computing, SOCC '14*, New York, NY, USA, pp. 14:1–14:14. ACM.

[27] Foster, I. and Kesselman, C. (1997) Globus: a metacomputing infrastructure toolkit. *Int. J. Supercomput. Appl.*, **11**, 115–128.

[28] Litzkow, M., Livny, M. and Mutka, M. (1988) Condor—A Hunter of Idle Workstations. *Proc. 8th Int. Conf. Distributed Computing Systems*, San Jose, CA, USA, June, pp. 104–111.

[29] Anderson, D.P. (2004) Boinc: A System for Public-Resource Computing and Storage. *GRID '04: Proc. 5th IEEE/ACM Int. Workshop on Grid Computing*, Washington, DC, USA, pp. 4–10. IEEE Computer Society.

[30] de Oliveira e Silva, J.N., Veiga, L. and Ferreira, P. (2008) Nuboinc: Boinc Extensions for Community Cycle Sharing. *2nd IEEE Int. Conf. Self-Adaptive and Self-Organizing Systems (3rd IEEE SELFMAN Workshop)*, Venice, Italy, October. IEEE.

[31] Andrade, N., Cirne, W., Brasileiro, F. and Roisenberg, P. (2003) Ourgrid: An Approach to Easily Assemble Grids with Equitable Resource Sharing. *Proc. 9th Workshop on Job Scheduling Strategies for Parallel Processing*, Seattle, WA, USA, June.

APPENDIX

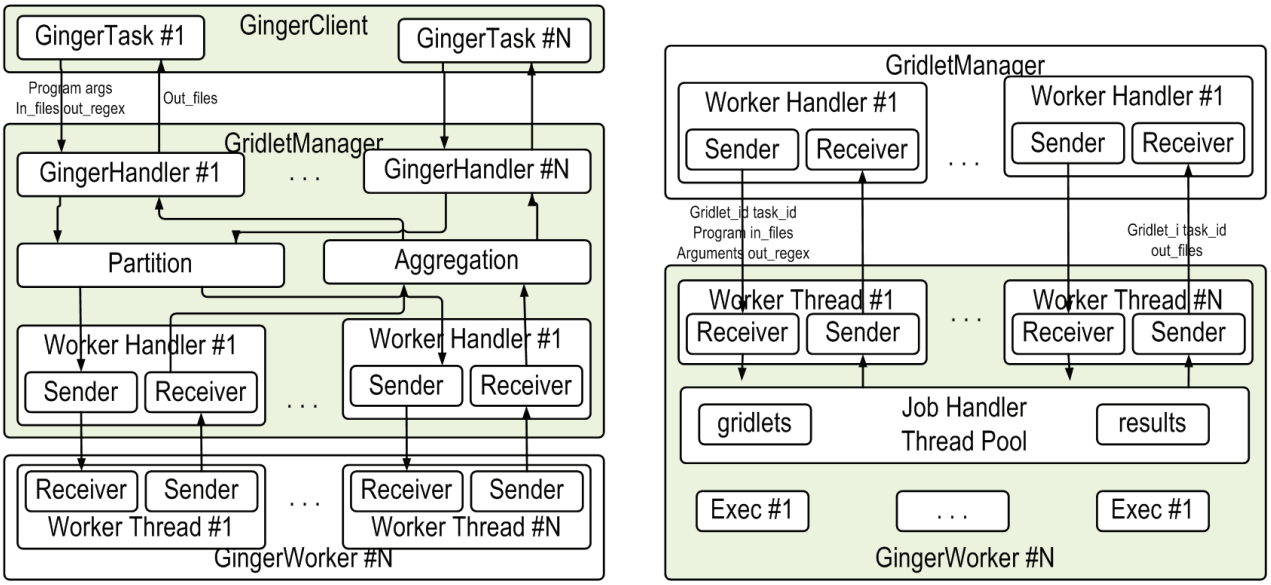

FIGURE A1. C^{3}P gridlet manager.

```
<all>
<!-- delete unnecessary frames and indexes -->
<variable name="rem_size" select="sum($start_frame/preceding-sibling::frame/@size)"/>
<delete context="$start_frame" select="preceding-sibling::frame"/>
<delete context="$end_frame"
select="following-sibling::frame"/>
<delete context="$start_index" select="preceding-sibling::index"/>
<delete context="$end_index"
select="following-sibling::index"/>

<!-- update indexes -->
<update select="/idx1/index/chunk-offset" value="@value - $rem_size"/>

<!-- update list sizes -->
<update select="/size" value="sum(/*/@size) - 8"/>
<update select="/movi/size" value="sum(/movi/*/@size) - 8"/>
<update select="/idx1/size" value="sum(/idx1/*/@size) - 8"/>

<!-- update frame counts -->
<variable name="vfcount" select="count(//frame[substring(chunk-id,3,1)=?d?])"/>
<variable name="afcount" select="count(//frame[substring(chunk-id,3,1)=?w?])"/>
<update select="//total-frames" value="$vfcount"/>
<update select="//strh[fccType=?vids?]/length" value="$vfcount"/>
<update select="//strh[fccType=?auds?]/length" value="<brace>afcount"/>
</all>
```

Listing 1. XML descriptor for AVI file partitioning.

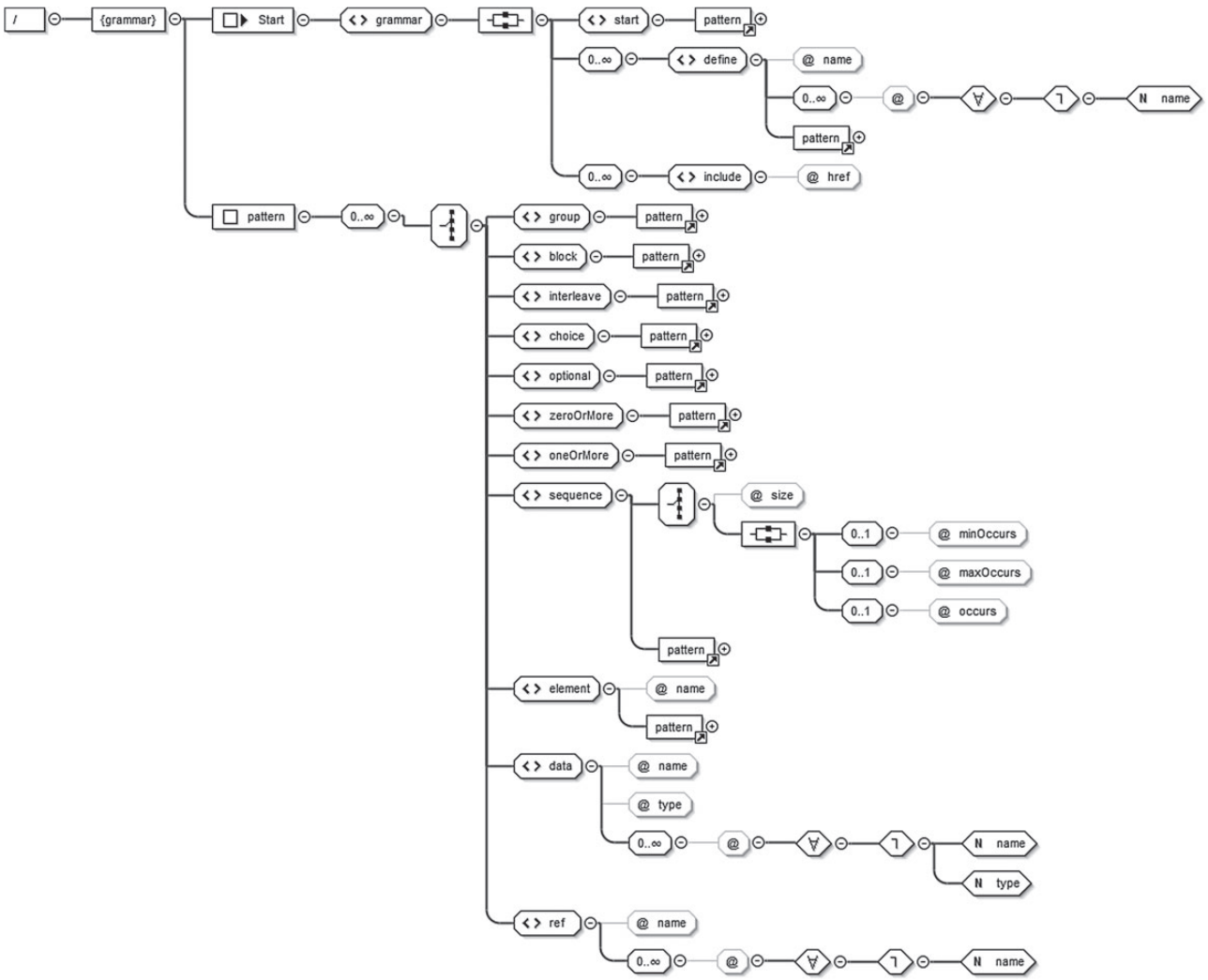

FIGURE A2. Grammar for format and application descriptors.

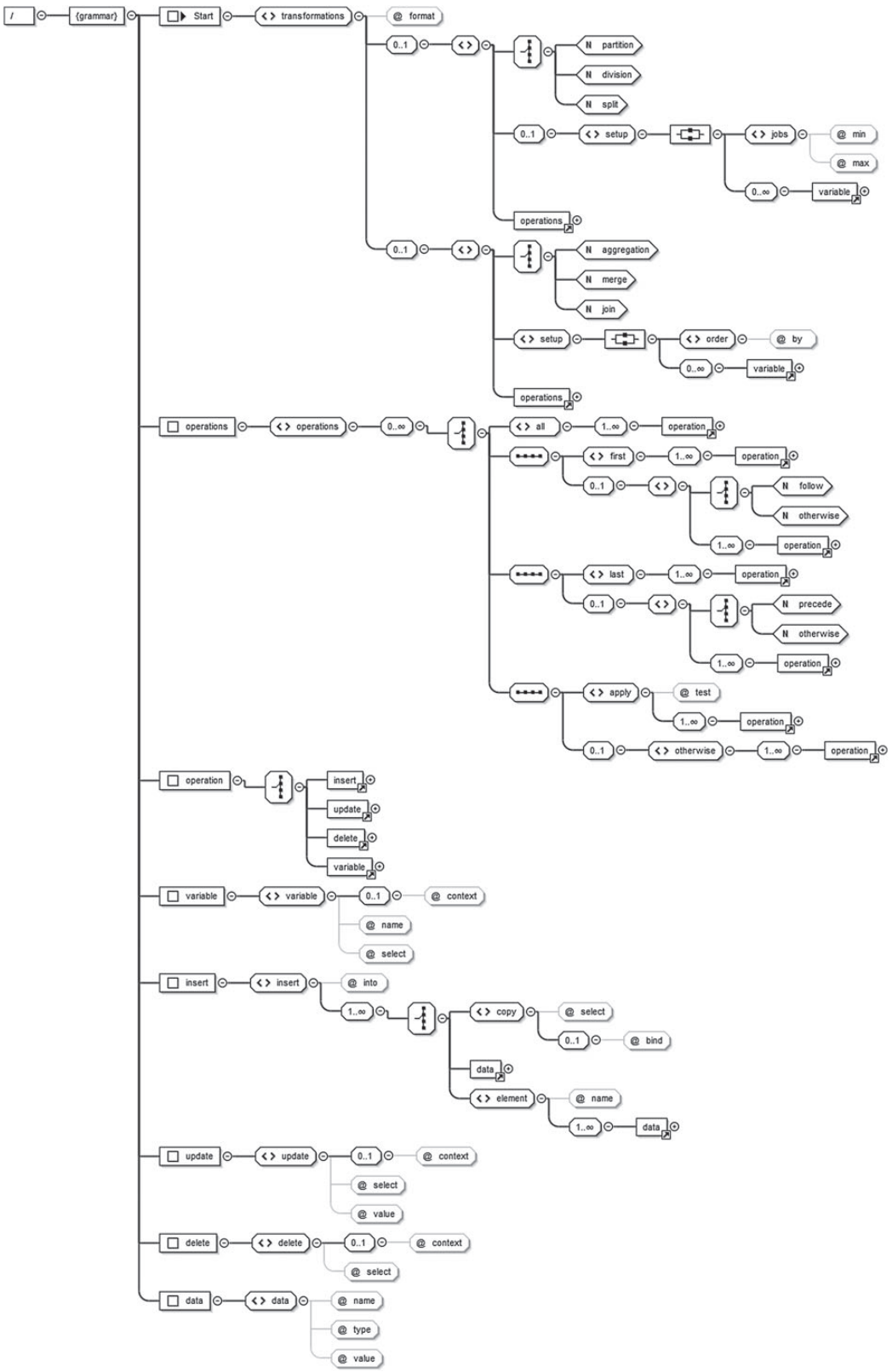

FIGURE A3. Grammar for partition and aggregation transformations.