*Next Generation Concurrent Engineering - M. Sobolewski & P. Ghodous (eds)*
*© 2005 ISPE, Inc., ISBN 0-9768246-0-4*

# Service Oriented Architectures and Grid computing – A New Generation of Applications for Grid Enabled Data Centers and Public Utility Computing

Heinz J. Schwarz
*Sun Microsystems, Inc., Menlo Park, CA, USA*

ABSTRACT: Cluster- and Grid infrastructures have gained significant popularity within the scientific community, and in recent years also became the platform of choice for commercial applications in various fields. We will argue in this paper that a combination of service oriented architecture style systems design, implemented within cluster- and grid infrastructures, and controlled by dynamic multi-agent systems, will enable new types of applications that can not only leverage the resources of datacenters most efficiently, but also form the foundation for public utility computing. Key to this technique is the decomposition of traditional applications into functional units, which are then exposed as services. We will describe the basic requirements and control mechanisms, as well as the benefits of this novel systems design.

## 1 DEFINITIONS AND FUNDAMENTAL TECHNOLOGY

### 1.1 *Definitions of Cluster- and Grid Infrastructures*

We define for the purposes of this paper a cluster as a collection of independent, homogeneous computational entities that are located in physical proximity, controlled by a single resource scheduler. A grid is defined as a collection of multiple resources that can be inhomogeneous, managed by multiple local resource schedulers and located either in physical proximity or remotely connected. A grid, however, allows access to its resources through a singular interface and provides user level code access to computational, data, visualization or other resources.

### 1.2 *Definitions of Service Oriented Architectures*

In a very broad and general definition, "a service-oriented architecture is essentially a collection of services. These services communicate with each other. The communication can involve either simple data passing or it could involve two or more services coordinating some activity. Some means of connecting services to each other is needed"[1]. A service is "a unit of work done by a service provider to achieve desired end results for a service consumer"[2], and finally "both provider and consumer are roles played by software agents on behalf of their owners".

Elements of a SOA are usually

- Services that implement a contained set of functionality, for example business logic or data operation, with a defined interface. The interface needs to be light-weight and implemented in a platform independent fashion, for example with Java, Jini or XML.
- A registry where all services publish their interfaces and expose their functionality. The registry is used by services to locate provider services.
- Clients, which can be other services, that locate a provider service through published interfaces in the registry and connect using the interface. (Adapted from Singh et al, 2004[3])

For the purposes of this paper it needs to be noted that a service could be fairly complex. The concept of a SOA does not require the service itself to be light weight, only its interface. The service can be implemented in a native method like C or FORTRAN, while the interface description needs to be platform independent. Exposing the functionality of a native method program to a public interface is called wrapping and the connector between the program and its public interface is called a wrapper. If a service is implemented in a object oriented language, an object can be invoked from its class, but a native method program could be started by the interface. One of the key advantages of SOA is the loose coupling of elements through the interface publishing and subscription method and the registry. *N* consumer services can connect to *M* provider services, which makes SOA inherently scalable. Agents are roles of services.

## 2 CLUSTER AND GRID INFRASTRUCTURE

### 2.1 *Cluster design parameters*

Clusters are usually built from homogeneous components. There are several good reasons why this makes sense. One set of reasons encompasses ease of administration, software distribution and handling. It is better to have two clusters with homogeneous components within them, than one cluster with heterogeneous components. This becomes drastically evident if nodes are provisioned with boot images, which has several advantages, but requires a homogeneous setup. Another reason to keep nodes homogeneous is application mapping. Given that a Datacenter provides resources with different architecture profiles, it makes sense to allocate resources that most efficiently match the application requirements. If a cluster is built from homogeneous components, it is easy to characterize.

In order to allow all cluster nodes to run concurrently different threads of a single application, it is necessary to create access to a single, shared file system. Access to data prior to job execution and during runtime can influence the execution performance significantly. Hence, the design of the shared cluster file system deserves special attention in terms of aggregate bandwidth and scalability. Latency is not as relevant given that data access from disk is several orders of magnitude slower than local access within memory. We assume that application programmers already optimized their applications in terms of data caching and pre-fetching. On the other hand, for parallel applications with huge amounts of communication, in this case specifically messages, it is often recommended and common practice to use low latency, high bandwidth interconnects. Some applications are more sensitive to interconnect latency than others, but the general principle is that the more communication is required between the nodes, the more communication bandwidth and latency becomes a bottleneck. The speed of an individual CPU can become less performance relevant than the communication characteristics of the cluster nodes.

In order to use a cluster as a singular resource, it is common practice to define one master node that keeps track of available resource within the cluster with a Dynamic Resource Management (DRM) system like Condor, PBS, LSF or Sun N1GE6[4]. Most of the common DRM systems collect jobs submitted for execution on the cluster in one or many queues and allocate resources to the job based on a policy or scoring mechanism. In the simplest form, a queue would just sequentially store and forward jobs (first-in, first out). More sophisticated systems allow policy management, advance reservation with backfilling and metering of queue and execution times per job, user, group and project. Every job is evaluated upon submission to the queue. This model creates a relatively static queue. Given a deep queue on one side and a dynamic system on the other side, many things can happen between initial job priority scoring and its actual allocation. For large, scalable environments, we will suggest a more dynamic model based on multi agent systems. However, some of the traditional DRM features can be exposed to higher level management tools through the GRAM interface. The GRAM interface within the Globus Toolkit[5], "provides a single interface for requesting and using remote system resources for the execution of 'jobs'. The most common use of GRAM is remote job submission and control. It is designed to provide a uniform, flexible interface to job scheduling systems". Higher level services can build on GRAM to allow job scheduling across a grid, as implemented for example in the grid-lab resource management system[6]. We described here only a few of the most important cluster design parameters. Other parameters like power consumption in relation to performance, cooling, physical layout etc. are not pertinent to the topic of the paper and are therefore omitted, which by no means suggests those parameters were less important.

### 2.2 *Grid Design Parameters*

According to our definition, we deal with grid design parameters already when we have more than one cluster and wish to allocate jobs, or as we will later define *decomposition blocks*, across those multiple clusters. In reality, we could have different clusters with homogeneous nodes among them, yet heterogeneous characteristics between the clusters, and a collection of different size single systems. We refer to size here as shorthand for characteristics like number of CPUs, memory, I/O bandwidth and so on. System ß shall be defined as bigger than system Δ, if ß has more of either of the aforementioned characteristics. In order to be able to access resources within the grid, the grid offers a singular interface, which could be either a log-in node with a public IP address or a portal server. The log-in server needs to record some basic user information and authenticate a user to check if she is authorized to submit jobs to the grid. We need all of these features for a single system or a single cluster also, but request that the grid middleware acts as a gateway, hence handling the authentication and authorization process once for all resources within the grid. In some grid environments, we find resource scheduling capability on a grid level, which is desirable. However, many current grid implementations require a user to select manually on which grid resource the application should be scheduled. Resource scheduling at a grid level requires a feature we will refer to as *meta scheduling*. Meta scheduling is a functionality that matches resource requests with resource availability of the grid resources. Every grid resource should have a local interface that collects resource information and can report those in an aggregated form to a higher level scheduler. GRAM is a basic interface definition that allows implementing *meta scheduling* across multiple grid resources. However, sophisticated features, like advance reservation and backfilling, which are certainly common in local resource management environments, are difficult to implement on this level, as GRAM defines only the exchange of basic information.

Another important consideration regarding grid design involves access and management of data. We defined that, in a cluster, all cluster nodes shall have access to a single file system simultaneously to allow the execution of a single, parallel application. In a grid environment, several different scenarios are conceivable. Architecturally difficult to implement, yet simplifying from an application programmer's point of view, would be a singular file system for all grid resources. In current grid implementations, applications usually do not use resources concurrently, but sequentially. An application might be submitted to a meta scheduler, scheduled for execution at one resource, executed and the result stored in a defined directory. In this scenario, a singular file system is not necessary. The data could be transferred along with the executable program to the designated grid resource and stored in a file system local to this grid resource. However, what needs to be considered in this case is the bandwidth available to transfer data between one grid resource and another. A large data transfer of several hundred Gigabytes could take a substantial time, which needs to be considered as resource utilization itself. With our previous example we have introduced the concept of simultaneous data access, which requires a single file system, and sequential data access, which requires data transfer between grid resources. We also defined the basis for a service requirement we will describe in detail later. Data transfer is *costly* in itself, as it requires considerable resources. A large data transfer requires I/O capacity on two servers (sender and receiver), network bandwidth and storage capacity. Moreover, data transfers are costly in terms of time consumption. We shall from hereon refer to *cost* as a description for the consumption of any measurable resource. In order to increase efficiency within the grid, we might have to deal with trade off decisions. Transferring data from one resource to another might make sense under the aspect that more appropriate CPU resources or Storage resources can be used at the receiving resource, but we incur cost for the data transfer. If the benefit of the transfer outweighs the cost, we should do it, but at the same time we shall avoid transactions if the cost outweighs the benefits. Most current grid middleware environments have no means to consider data transfer cost while making resource allocations, if resource allocation are made automatically or dynamically at all.

## 3 APPLICATION DECOMPOSITION

### 3.1 *Native Methods and application decomposition*

Functional application decomposition is a method fairly common in high performance computing environments. We will define basic elements with a simple example depicted in figure 1. It is fairly common that a single application has different sequential phases, irregardless of the actual implementation with a native method programming language. Let's assume that our sample application requires some sort of pre processing, for example sorting and ordering input parameters. In the second phase, the application accesses a large database based on the preprocessed data. The database access requires several complex join operations between various tables. This access populates a sparse matrix, which then can be processed in an iterative loop. Every step of the iteration performs a simple operation on the data.

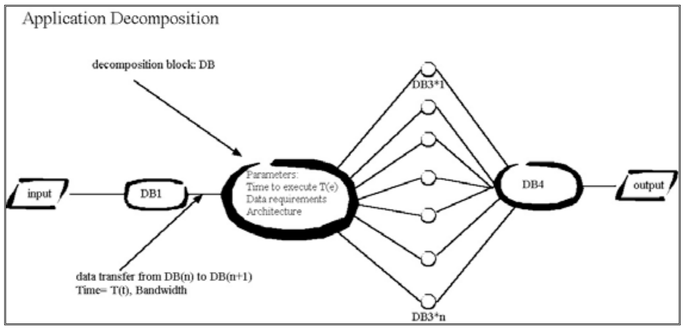

**Figure 1**

In the next step, all the results are combined in a post processing operation and written in an output file.

Each of the previously described phases of the program requires a different set of resource characteristics and a set of operations. Traditional decomposition is interested in looking at these groups to identify sequential dependencies and potential parallelization targets to speed up the execution of the application. However, in our example, one phase is data dependent on the previous phase. Optimizations can occur within the phase. We will refer to these sequential phases with defined, common characteristics performing a set of defined functions from now on as *decomposition blocks*, indicating that we decomposed the application into sequential blocks, each of which operating a set of functions. In a traditional program, the sum of all decomposition blocks equals the sum of all operations of the program and hence of the application. Another way to describe a monolithic application would be to call it the aggregate execution of decomposition blocks. In our example, we label the decomposition blocks sequentially DB1 to DB4. For the purposes of our concept we shall now address two other characteristics: the execution time of the decomposition block, and both the incoming and outgoing data volume. As we move from $DB_n$ to $DB_{n+1}$, we have to pass the output data of $DB_n$ forward as the input data of $DB_{n+1}$. In a trivial case, the entire application with all decomposition blocks is executed in one single computer system with shared memory, so the data transfer is nothing more than data handles defined within the program pointing to the same logical storage pointers, which could be either in cache, memory or the file system. The operating system handles the allocation between the logical and physical representation. In cluster and grid environments, data handling is more complex.

We measure the execution time of one decomposition block as the time to access input data, perform all operations of the decomposition block and write the output data. Therefore, the execution time is not only dependent on the compute time, but also on the time to transfer data, which in turn is dependent on the bandwidth and data volume. In a simplified form we define $T_t = T_e + T_d$ , in which $T_t$ is the total runtime for a decomposition block, $T_e$ the execution time and $T_d$ the time required for data transfer.

Another concept we need to define in our model is related to parallel execution. Some decomposition blocks, like DB3 in our example, allow parallel execution. As we perform the operations of the matrix, each operation is not dependent on another operation, only on the input data in relation to an index *i*. Therefore, we have actually *n* implementations of DB3 that are index dependent on *i*. We shall define each parallel executable functional unit of a decomposition block as a parallel decomposition sub-block (PBSB). In our example, DB3 can be executed in parallel with *n* PDSBs, in which *n* is dependent on *i*. We refer to each PDSB as an instantiation or copy of the DB with the index i and label accordingly. The execution time for DB3 is the aggregate of all PDSB. DB3 is an ideal typical parallel DB which requires little or no communication between the nodes executing each PDSB. It is not uncommon to find DBs that can be represented in PDSBs, yet the calculations are somewhat dependent on each other and require the exchange of data. Data exchange within PDSB is often referred to as messages.

We could combine all decomposition blocks into one application and run them on different architectural platforms, as is common practice today. In that scenario we will encounter trade off situations. DB1, for example, is a fairly simple decomposition block with a small data input and output set and no spectacular requirements in terms of CPU or memory requirements. DB2, however, has a different set of characteristics. We can execute it on a large SMP system, which would allow us to perform the join operations within memory, which will speed up the execution time significantly. If we decide to execute the decomposition block on a cluster, we will likely exceed the memory of the individual cluster nodes and have to rely on a parallel database access system, such as Oracle 10G. However, we will experience heavy data exchange between the cluster nodes, as the join operation requires communication between all elements. On the other hand, DB3 is an ideal set of operations for a cluster environment, as all operations can be performed in parallel.

On a single SMP, the OS will treat different PDSBs as threads. If we have a sufficient number of CPUs or CPU equivalent cores available to execute the PDSBs, each thread will have its own fast cache memory to speed up memory transactions. Message exchange between the PDSBs, if required, could use the relatively fast memory system, which should reduce the network cost compared to an external network required in a cluster. In case of an oversubscribed system however (i.e. more PDSBs than CPUs), caches will be shared between threads. This will lead to cache contention, in which case one thread will evict data that is still needed by another thread. Such a behavior is costly as it requires reloading data numerous times. One way to address this costly behavior would be to match the size of the SMP environment to the number of parallel threads. While this reduces the overall cost of running a parallel application on a SMP system, the cost of a large SMP system for running this type of application might be high[7]. We will discuss the difference between the cost of an application running on a system and the cost of a system running an application hereafter and in section 4.2.

We have given so far just some rough examples, based on heuristics, to demonstrate different types of decomposition block profiles. Current practice is to optimize an application as a collection of decomposition blocks and select one architecture target. As we have demonstrated, any architecture can have advantages and disadvantages for any decomposition block. We could take the entire application and execute it on different platforms and define that the platform with the best overall execution time would become the preferred target platform, the one with the second best time would become the second priority and so on and so forth. The result would be static and could be stored in a scoring table. We suggest, in contrast, a three step process.

In the first step, we perform traditional application decomposition and optimize each decomposition block. In the second step, we measure $T_e$ for the entire application on each of the available architectures and build a scoring table. We shall refer to the execution of a collection of decomposition blocks further on as aggregate execution. The score provides a ranking of architectures based on the cost of aggregate execution. Then, in a third step, we perform the same scoring for each decomposition block separately on different platforms. We shall refer further on to the execution of decomposition blocks on different systems as disaggregate execution. If we achieve a higher score (lower cost) with disaggregate execution, we suggest exposing each decomposition block as a service and using different resources within a grid infrastructure to execute the application not as an aggregate of decomposition blocks, but disaggregated. The better the bandwidth between the grid resources, the lower will be $T_d$ and hence the likelihood that a disaggregated model achieves a higher score than an aggregate model. If $T_d$ was equal to zero, which would equate to zero time delay for data transfer, every decomposition block could be executed on the platform that is most appropriate and efficient.

## 4 IMPLEMENTATION

In the previous sections of this paper we described design parameters for clusters and a grid infrastructures built from multiple clusters and other resources. We also described a process called application decomposition that analyzes functional blocks within an application. In the next two sections we will describe how these decomposition blocks are turned into services and mapped to the various potential resources in a grid environment.

### 4.1 *Exposing Interfaces of decomposition blocks*

We defined earlier a service as "a unit of work done by a service provider to achieve desired end results for a service consumer"[8]. With the process of application decomposition we created a set of service providers. Each service provider or decomposition block has specific characteristics. We can describe these characteristics in a standardized way and attach these descriptions to an interface layer for the service provider. The interface layer also describes input and output parameters. The combination of the service provider and its interface layer is called a *service*. We can use an instantiation of the interface layer to represent the service in a service directory, which allows the agents of other services to locate and connect to the service. Thereby, we have all the essential elements of a SOA, namely services, a registry and clients that connect to and use the services.

### 4.2 *Mapping of decomposition blocks with service descriptors*

As we defined already, a decomposition block is a set of functions with defined, common characteristics. One of the implications is that we can test the performance of each decomposition block in a coherent way, as the definition of the decomposition block required common characteristics. All functions performed within a decomposition block require similar access to resources, be it integer or floating point threads, memory access, communication etc. We can measure and describe all of these resource requirements and determine the cost specific to different platforms. The best fit of a decomposition block to a specific architecture is the architecture that incurs the lowest cost (see figure 2).

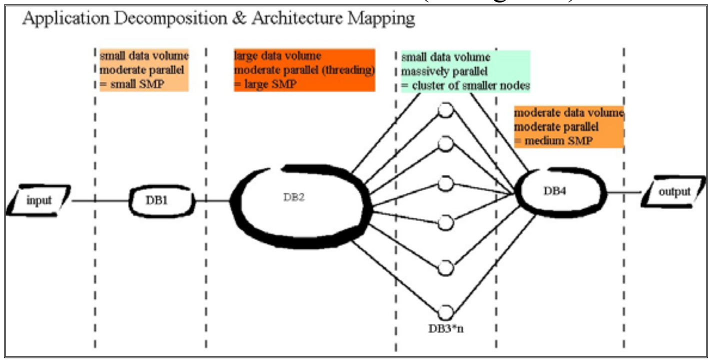

**Figure 2**

We can measure for example time, memory consumption, bandwidth consumption and compute thread consumption on different platforms available within a grid to determine the best fit. In our sample application, DB2 might not need a lot of compute threads, but it needs a lot of memory and communication. Implemented on a cluster, we would still have the same consumption of compute threads, but we'd now need additionally a lot of I/O threads to enable communication which is less time consuming within an SMP system. Communication also creates CPU overhead, resulting in increased cost. At this point, we do not correlate the cost of executing a service on a specific platform, and the cost of a platform executing a service. Instead, we create a scoring table that determines architecture match based on lowest overall resource utilization. This architecture related cost is static and therefore a fixed cost. The cost of a platform running a service, in contrast, is not fixed. It can depend on different factors and time. Cost calculation for a platform consists of different types of costs, for example capital costs (acquisition, depreciation), operating costs (power, cooling, floor space, services etc.), connectivity costs (bandwidth of the connection, bandwidth related costs). Most of these costs are relatively fixed, which leads to the quick conclusion that they determine the price of a service executed on this system. But this is a false assumption. It might be that the cost for one CPU/hour in a cluster is \$1, whereas the cost of one CPU/hour within an SMP system is \$5. It is furthermore reasonable to assume that this cost remains similar or equal, regardless of actual utilization. Energy utilization actually varies not only with the level of utilization, but varies with the very nature of a specific application. But for the purpose of our discussion, we assume fixed energy consumption. Under these circumstances, it would make sense for the owner of the resource to make sure that the resource is utilized close to or at capacity, as that would lower the price per unit of work. If its reasonable to assume that the cluster will be utilized at

40% of capacity, the price of a unit (for example CPU/hour) must equal at least the cost at that capacity. Incremental utilization would lead to marginal revenue. We define this point as the break even point. In the example, the break even point would be reached at 40% utilization with a return of $1 per CPU per hour. The break even point for a system could be calculated at a higher or lower utilization, based on a policy decision of the resources owner. Resource owners can implement more aggressive or less aggressive strategies to utilize their resources and maximize marginal revenue. It would be conceivable to drop the price close to or to marginal cost in periods of low utilization, for example at night or on weekends. Since marginal cost (the cost for an additional work unit) is practically zero, the price for a marginal unit of computing could be set to just $0.50. We are not suggesting that the cost of an SMP system is lower than the cost of a cluster. What we suggest is that the marginal cost for a work unit is small or zero and therefore marginal prices for compute resources can be set policy based. As a consequence, the allocation of a decomposition block needs to be calculated based on variable factors, not merely on platform resource utilization.

| ***Platform*** | ***Cost on Platform*** | ***Cost of Platform*** | ***Marginal Cost of Platform*** |
|---|---|---|---|
| ***A (SMP)*** | 10 CPU h | $5 | $0.50 |
| ***B (small SMP)*** | 20 CPU h | $2 | $0.50 |
| ***C (cluster)*** | 40 CPU h | $1 | $0.50 |

Table 1 – cost/platform relationship

Table 1 represents a simple example to demonstrate the consequences of this observation. We assume the table represents the cost of DB2, expressed in CPU hours, running on three different platforms. The resource utilization could be easily calculated at $50 on platform A, $40 on platform B and also $40 on platform C. If our optimization goal was shortest runtime, the lowest cost would be provided by platform A. If the optimization goal was lowest expense, platform B and C would be equal choices. In that case, availability of the platform, namely queue depth, could influence the platform decision. If all platforms are offered at marginal cost, because their target capacity had been reached, platform A becomes the most attractive platform, because it incurs the lowest cost and the lowest expense.

So far we referred to the cost of a DB running on a particular platform as a fixed cost. We need to clarify that even this cost varies with the data set. The same service and can operate with different datasets, and the dataset will determine the total cost for the execution of a service on a platform, as the cost of handling data can vary by platforms. Therefore, we need to specify rules that allow to calculated, how different data sizes impact the cost of execution of a service.

The purpose of the previous discussion is to show the difference between the cost of running an application on a platform, which is a fixed relationship, and the cost of a platform running an application, which is variable. Instead of implying a result, we suggest to define a framework that allows ad hoc platform allocation based on rules and policies.

Service descriptors define the characteristics and platform requirements of a service, but they should not map a service to a particular platform. Instead, service descriptors provide information and rules that allow other services to perform ad hoc mapping services, based on an evaluation of conditions within the grid infrastructure at the time of the mapping. The mapping service should continuously collect information about availability of resources and the marginal cost of utilization, similar to the function of a DRM in a cluster environment.

## 4.3 *Policy implementation*

We established that cluster and grid environments are dynamic, non deterministic systems. We suggest that the connection between services should leverage and adjust to this dynamic system. Instead of determining invocation and utilization of services or mapping of service to a platform upfront, we suggest to implement a rule based framework, using a multi-agent system (MAS) control structure. "The characteristics of MASs are that (1) each agent has incomplete information or capabilities for solving the problem and, thus, has a limited viewpoint; (2) there is no system global control; (3) data are decentralized; and (4) computation is asynchronous"[9]. Whenever one service provider (decomposition block) concludes the work of its designated set of functionality's, services are invoked to determine the best possible execution platform for the next service provider. Parameters to consider are the cost of execution on platforms available within the grid, cost of data transfer and the cost of the platform. All those parameters change dynamically. In order to determine cost of data transfer, available bandwidth, data volume and network access, data transfer costs have to be evaluated by a data transfer agent. Every platform can advertise a price for resource utilization, which could be based on a full or partial return calculation with marginal cost incentives. This function could be performed by a pricing agent, which retrieves information from single systems, DRMs of clusters or equivalent grid interfaces. A scoring agent with information about the architecture scoring can evaluate the information provided by the data transfer agent and pricing agent to determine, what at any given time the best execution platform would be. Users can determine if they want decisions to be optimized for shortest time, or lowest cost. This decision impacts the decision making at run time. Additionally, resource owners define policies that determine how, to whom and with which policy to advertise their resources and which workload to accept. A multi agent based policy framework creates a dynamic, self adjusting control system, in which the optimization strategies are rule based, with every decision made in response to current conditions.

## 5 RELATED WORK

SORCER at Texas Tech[10] is related, as it creates a network of services for existing applications. The implementation uses JINI and Java, wrapping existing applications and exposing them as a service. Gridlab[11] is a related architecture using existing grid middleware while adding advanced services. Both projects contribute an extensive body of knowledge and sets of sophisticated tools. John McClain's[12] work on building scalable and adaptive systems using Java and Jini technology and the RIO project[13], are related. RIO provides a framework for dynamic provisioning and the idea of dynamic containers.

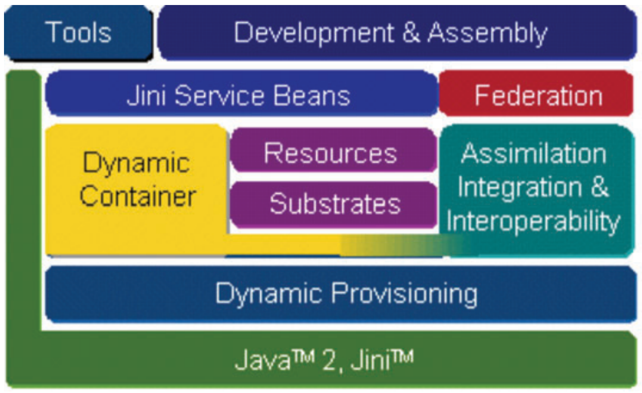

**Figure 3** Rio Architecture

Rio introduces the concept of dynamic containers, which is very important in the context of this paper. "The Dynamic Container support in Rio is called a Cybernode. Cybernodes embrace the recognition that the network is composed of heterogeneous compute resources with multiple architectures, operating systems - all with different capabilities. The Cybernode provides a lightweight dynamic 'agent' turning heterogeneous compute resources into services available through the network". It appears that RIO seems to be a very capable platform in the context of SOA implementations in grid environments.
All of those related projects deserve more consideration in a separate paper.

## 6 TOWARDS PUBLIC UTILITY COMPUTING

Within a single data center, we can address many of the issues of disaggregated execution with a control structure of multi agent systems. Architecturally, we can reduce data transfer times by providing high bandwidth interconnects between grid resources. A variety of platforms increases the likelihood of optimal matching. This structure forms also the basis for public utility computing. If an architecture that would be optimal for the execution of a particular decomposition block is not available in a data center, we could use agents to locate remote services and use the same scoring and evaluation agents to determine if the remote execution would be the lowest cost alternative. If the best possible environment to execute a decomposition block was, for example a 1024 node cluster, which we don't have available, the evaluation of our MASs could indicated that it was still better to transfer both data and code to a retail cluster (a cluster that offers compute cycles on demand), than to run it locally. However, using a public infrastructure requires several additional services for security, authentication and authorization, accounting and data handling. But the service oriented architecture implemented on clusters and grids with a multi agent control systems is fundamentally scalable, dynamic and extendable enough to master these requirements.

## 7 SUMMARY

In this paper we have discussed how the principles of SOA can be used to maximize the efficiency of resource utilization within a datacenter. Instead of compromising different elements of a monolithic application, here called decomposition blocks, into a single architecture at run time, we describe a model that will allow distributing decomposition blocks across different architectural platforms within a datacenter dynamically at run time. We combine the traditional method of application and data decomposition with exposing the functionality of the decomposition block through interfaces that are published in a registry, hence leveraging the advantages of SOA for data- and compute intensive applications realized in native methods.

Rather than coupling services or decomposition blocks statically with web services, we suggest a multi agent system that dynamically allocates the best possible execution platform based on rules.

We pointed out limitations of the system based on the cost of data transfer and decision making overhead. The granularity of the decomposition blocks has to be appropriate to yield benefits from this concept. Future research will determine the exact relationship between network, platform and application parameters to achieve those potential benefits.

## 8 ACKNOWLEDGMENTS

Albeit this paper represents the beginning of a research undertaking, not a final result, it has already been shaped by numerous vivid discussions. I'd like to acknowledge specifically my colleagues Victoria Livschitz and Ruud van der Pas. Victoria pointed me in the direction of multi agent systems to resolve the problem of using multiple concurrent optimization parameters for decision making in dynamic systems. Ruud is a first class expert in application analysis and tuning and provided invaluable input on section 3.

## 9 REFERENCES

W3C, Mapping of W3C Web Service Architecture work to SOA RM work, online at http://www.oasis-open.org/apps/group_public/download.php/12209/Mapping%20of%20W3C%20Web%20Service%20Architecture%20work%20to%20SOA%20RM%20work_05-04.pdf

[1] Service-oriented architecture (SOA) definition, online at http://www.service-architecture.com/web-services/articles/service-oriented_architecture_soa_definition.html

[2] He, Hao (2003), What is Service Oriented Architecture?, online at http://webservices.xml.com/pub/a/ws/2003/09/30/soa.html

[3] Singh, Inderjeet; Brydon, Sean; Murray, Greg; Ramachandran, Vijay Ramachandran; Violleau, Thierry; Stearns, Beth: (2004), Designing Web Services with the J2EE 1.4 Platform, Addison-Wesley, Boston

[4] The open source version is free and known as Grid Engine, online at http://gridengine.sunsource.net/download.html

[5] The Globus Toolkit, GRAM manual, online at http://www-unix.globus.org/toolkit/docs/3.2/gram/ws/

[6] Nabrzyski, Jarek, GRMS, online at http://www.globusworld.org/program/abstract.php?id=92

[7] This paragraph is based on contributions by Ruud van der Pas, Sun Microsystems Inc.

[8] He, Hao (2003), What is Service Oriented Architecture?, online at http://webservices.xml.com/pub/a/ws/2003/09/30/soa.html

[9] Sycara, Katia P; (1998), Multiagent Systems, online at http://www.aaai.org/Resources/Papers/AIMag19-02-007.pdf

[10] Sobolewski, Michael; [PI] online at http://www.cs.ttu.edu/~sorcer/about/about.html

[11] Online at http://www.gridlab.org/

[12] McClain, John; online at http://www.javasig.com/Archive/lectures/JavaSIG-JiniSunTechDays-JMcClain.pdf

[13] Rio project home, online at http://rio.jini.org/