The JoSchKa System: Organic Job Distribution in Heterogeneous and Unreliable Environments

Matthias Bonn[1], Hartmut Schmeck[2]

[2]Institute AIFB
[1]Steinbuch Center for Computing
Karlsruhe Institute of Technology – KIT –, 76128 Karlsruhe, Germany
{Matthias.Bonn, Hartmut.Schmeck}@kit.edu

Abstract. This paper describes a job distribution system which focuses on standard desktop worker nodes in inhomogeneous and unreliable environments. The system is suited for general purpose usage and supports both batch jobs and object-oriented interactive applications using standard Internet technologies. Advanced scheduling methods minimize the total execution time and improve execution efficiency, specialized to deal with unreliable failing worker nodes.

Keywords: Distributed computing, scheduling, monitoring, web services, desktop grids, parallel applications, organic computing, cloud computing.

1 Introduction

During the last 15 years, Personal Computers (PCs) spread more and more. Being irreplaceable in office and university service, they are also widely used in home areas. Today, a standard PC is equipped with a multi-core multi-GHz CPU and at least 2–4 GB of main memory. When used for its common dedication – office applications, email or web surfing – this power remains unused, because the box nearly all the time waits for user input.

In the course of the ubiquitous Internet almost every PC is connected to, many projects were built to utilize this idle time, BOINC [1] or zetaGrid [2], for example. To build up a local area desktop grid, there exist systems like Condor [3] or Alchemi [4] sometimes using flexible algorithms to distribute individual applications to a pool or desktop PCs. But they are not necessarily suitable to utilize the idle time of normal Internet-connected PCs, and, in general, not programming language independent.

The idea of JoSchKa (Job Scheduling Karlsruhe) [5] is to distribute the jobs in a request/response way: A central managing server has knowledge of about all jobs, while small autonomous background applications running on the worker-PCs query for them, similar to the tuple-spaces in [6]. When asked for a job, the server uses sophisticated (and in the scope of public resource/desktop computing unique) methods to select one suitable job matching the agent's characteristics, and responds to the agent what has to be done. The agent then has to download all specified files, execute the given command and transfer all result files back to the server (Fig. 1).

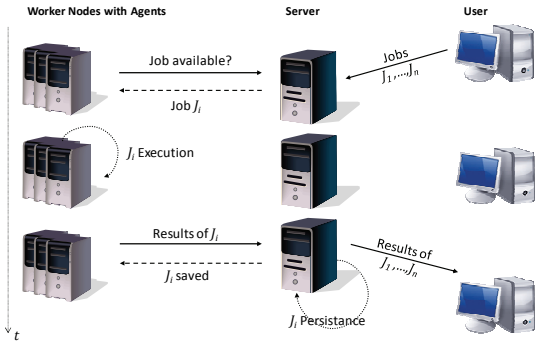


Fig. 1 JoSchKa principle

Typical features of JoSchKa are fully autonomous agents running on heterogeneous worker nodes and the support for any programming language (it has to be executable on the worker node, of course). The management communication and even the file transfer rely on HTTP only, there is no common file system. So the nodes can be run behind firewalls and NAT routers without problems. To support the users, there are both a graphical user interface for batch job upload and management, and a modern object-oriented API to develop interactive parallel applications.

In the following chapter, the server and its two mainly used distribution methods are described briefly. In the next sections, the agent and the user interface are described shortly, followed by its relation to organic computing and some practical experiences. The paper finishes with an outlook related to cloud computing.

2 Server

The server is the central component in the JoSchKa system. It has to deal with the up-/download of files, the management of the jobs and the decision, which one to select for a node if the agent running on this node polls for a job.

The file transfer interfaces used by the agents are standard HTTP-based interfaces, the ones for the user are SMB-based, due to convenient drag and drop file transfer. The most complex interface is the interface used to exchange management and control data, perform job queries, upload job descriptions and so on. This interface is a SOAP-based webservice [7] using HTTP as transfer protocol.

The agents on the nodes address the HTTP-interfaces only. So they can be run behind firewalls and NAT routers. As a result, the server is not able to distribute the jobs or send other management data to the agents using a pushing-style but it has to react on queries by the agents and send commands as a response to these queries. If the server wants to tell something to an agent (*"execute this job:..."*), it has to wait for the agent performing a request for exactly this information (*"job available?"*). If an agent queries for a job, the server first creates a candidate list containing all jobs which are able to be run on the node. To this, the server performs some checks with

every job in the database (the server manages a record of 22 values per job; the detailed description of the data model is omitted here):

- Do the platform specifications of the job and the agent match (operation system or memory requirements, for example)?
- Does the type of the job match with the type queried by the agent?
- All specified source files are physically available for download?
- If the job has predecessors defined, are they all finished?

If all tests evaluate to true, the job is added to a candidate list from which the scheduler has to select one single job. But before describing the selection process, it is necessary to explain why it is even a problem to select a suitable job and why not just selecting any job. Many publications (like [8, 9]) are dealing with intelligent scheduling algorithms, but they all are running on a reliable cluster or a super computer and generally have full knowledge about the nodes and the jobs' runtime requirements. None of the common desktop job distribution systems like BOINC, Condor or Alchemi try to observe the nodes and to exploit this additional knowledge, especially to improve the total system performance. Here, we have to distribute jobs to heterogeneous, unpredictably failing nodes. So other methods are needed, especially a monitoring of the nodes is essential. These methods are described in the following sections.

2.1 Fault Tolerance

When using a dedicated cluster, one can be sure that every node executes its jobs without failing before the current running job is finished. But when the nodes are part of an unsecure pool or if they are used as standard desktop computers we have to face with failures and unfinished jobs, because users normally shut down or reboot the PCs. So there are some mechanisms integrated to counteract this problem:

- The upload of intermediary results from the agent to the server and
- a heartbeat signal from the agent to the server.

The user has to activate the intermediary upload explicitly. If active, the agent uploads all new or changed result files in periodic intervals. So one can decide if a job has to be stopped or – in case of a node failure – started again. But this decision has to be made by the user and, of course, depends on the characteristics of the problem the jobs are working on. If the user does not block a failed job explicitly, the server restarts it after a certain time period. For every job, the server manages an internal counter, which is periodically decreased by 1. Every time the agent which executes the job sends the periodic heartbeat signal, the counter is set back to its maximum value. If the node with the agent shuts down, the heartbeat for this job will be missing and the counter reaches zero. Then the server creates a new ID for the job and marks it available for a new run. Due to programming mistakes or other software errors it may happen that jobs fail even on reliable nodes. A special mechanism takes care of restarting only for a finite number of times. Jobs which are failing too often are blocked automatically by the server until the user fixes the problem and removes the lock manually. The acceptable number of failures until auto-blocking can be changed by the user.

2.2 Node monitoring

A central assumption during the design of JoSchKa was the heterogeneity of the nodes. They not only differ with respect to their hard- and software configuration, but also in their reliability characteristics. Some nodes process every job successfully but others break down in an unpredictable manner. They may also change their reliability or switch from unreliable to robust and vice versa. Imagine a PC in a student pool, which runs nonstop for many hours in the night, but gets rebooted very often at daytime, when students use this PC.

To assess the behavior of the nodes a monitoring component was developed. The goal is to reach the ability to predict the behavior of the node in the near future [5]. Amongst others, the following values are monitored and calculated for each node, respectively:

- B: The relative CPU power of the node compared to the others, $B \in \mathbb{R}^+$.
- avU: The average uptime in wall clock minutes, $avU \in \mathbb{N}$.
- acU: The current uptime in wall clock minutes, $acU \in \mathbb{N}$.
- R: A synthetic reliability index of the node, $R \in [-1,1]$.

In the following, let $EWA(v_n, \dots, v_1)$ be the exponential weighted average, which processes the single input values v_i considering their age. It is defined recursively: $EWA(v_n, \dots, v_1) = \tilde{v}_n = \alpha v_n + (1-\alpha)\tilde{v}_{n-1}$ with $\alpha \in [0,1]$ and $\tilde{v}_1 = v_1$. The v_i are sorted by age, v_n is the youngest. More formally, the values for a single node $i \in \{1, \dots, l\}$ are calculated as follows:

- $B_i = \frac{\sum_{j=1}^{l} rB_j}{l \cdot rB_i}$, where rB_i denotes the time span (in wall clock milliseconds) the node needs to process a fixed arithmetic-logical benchmark. This benchmark is done by the node in periodic intervals.
- $avU_i = EWA(u_{t_u}, \dots, u_1)$, where the u_j are the node's last t_u uptimes.
- $acU_i = T_{now} - T_i$, where T_{now} denotes the current wall clock time and T_i denotes the starting wall clock time of the node.
- $R_i = EWA(r_{t_r}, .., r_1)$, where the r_j are reliability criteria for the last t_r processed jobs. If a job is done successfully, then $r_j = 1$, if failed $r_j = -1$.

In practice, we chose $t_u = t_r = 10$ and $\alpha = 0.25$. These data which is collected by the server for each single node can be summarized as follows: For each node, we now know,

- how fast it can work compared to all other nodes (B),
- how long it is available (on average) before it breaks down (avU),
- how long it is up since its last boot (acU) and
- how reliable it behaved when processing the last jobs (R).

This knowledge is used by the different distribution strategies, which are described in the following section.

2.3 Basic Job Distribution Strategies

As pointed out, the JoSchKa system is not designed for specialized clusters (although it can be run there too, of course), but to distribute jobs within a pool of strongly varying nodes. The nodes differ in performance but mainly they differ in their reliability. It is possible that a node works perfectly reliable for many days, but gets suddenly rebooted repeatedly and unpredictably. Then, long running jobs can't be processed any more on such a node. On the other hand, it would be a waste of valuable uptime, if a reliable node just processes jobs which are finished after a few minutes. So, if the distribution system disposes of many jobs which belong to many different users, it's obvious to distribute the jobs in an intelligent way. The goal is to minimize the breakdown-caused waste of processing time and to be concerned with fairness to the users, too. No user should feel disadvantaged.

At first, we describe the basic distribution strategies, in the following section (simulation) they are evaluated. JoSchKa uses various (combined) strategies, for lack of space we only show two of them here: The simplest algorithm and the one which normally is used in practice.

(a) Balanced (fair) distribution: Every user gets the same share of nodes to process his jobs.

(b) Uptime-based distribution: The decision, which job is selected for a polling agent is based on the agent's uptime behavior.

Normally, the scheduler does not differentiate between individual users, but single *JobTypes*. A job type is a user-defined subset of his jobs. So the different needs of an individual user who has his jobs grouped to probably more than one job type can be satisfied much better. If the number of known agents is less than the number of types, the scheduler does differentiate only the users, the different job types of the same user are aggregated to guarantee a fair distribution. In the following it is assumed that the system has knowledge of about l nodes and n jobs to be processed. The jobs belong to m individual types or users, respectively. Let $l, n, m \in \mathbb{N}$ and $m \leq n$. Furthermore

- J_i, $i \in \{1, \dots, n\}$ denotes job i,
- JT_k, $k \in \{1, \dots, m\}$ denotes job type k (or user k, if $l < m$),
- $jT: \{1, \dots, n\} \to \{1, \dots, m\}$ with $jT(i) = k$ denotes the type index of job i and
- avT_k denotes the (exponentially weighted) average runtime of the finished jobs belonging to type JT_k, measured in wall clock minutes.

Balanced Distribution

Let $nrWORKING_k \in \mathbb{N}$ be the number of jobs of type JT_k, which are running when a node asks for a job. The scheduler selects the job i which is minimizing $nrWORKING_{jT(i)}$. In other words, a node always gets a job of the type with the least jobs running.

As a result, at any time every type/user gets the same number of nodes to work for it/him. This fairness only shifts when a user has not enough unprocessed jobs to keep the balance.

Uptime-based Distribution

As the name of this strategy suggests, this selection algorithm mainly considers the uptime values of the nodes. In practice, a typical node is available for a certain time period, but then it gets shut down with increasing probability. This proceeds every day in the same (or similar) manner. The values avU_i and acU_i, which are determined for each node i help to predict how long an asking node will still be available. From these values, a further value, $avTARGET$, is calculated. The closer the current uptime gets to the average uptime, the shorter the jobs are which have to be selected. If the node's uptime exceeds its average uptime, the scheduler little by little selects longer-running jobs, depending on the node's total reliability. To account for the possible different CPU performances among the nodes, the relative benchmark index is finally integrated to scale the average runtime. In other words, $avTARGET$ estimates how long the average runtime avT_k of a job type k maximally should be, to get a job of this type successfully done by the considered node before it becomes unavailable.

$$avTARGET' = \begin{cases} |avU_i - acU_i|, & acU_i \leq avU_i \\ (R_i + 1) \cdot |acU_i - avU_i|, & acU_i > avU_i \end{cases} \quad \text{with } R_i \in [-1,1]$$

$$avTARGET = avTARGET' \cdot B_i$$

To estimate, which job type has to be selected, the average type runtimes are sorted (having $avT_i < avT_j, i < j$) and the corresponding mean values are calculated:

$$avTM_k = \tfrac{1}{2}(avT_k + avT_{k+1})$$

Finally, the ultimate run length target value $RLTV$ is calculated:

$$RLTV = RLTV' + rnd(-2,2) \quad \text{where}$$

$$RLTV' = \begin{cases} avT^*, & |avTARGET - avT^*| < |avTARGET - avTM^*| \\ avTM^*, & |avTARGET - avT^*| \geq |avTARGET - avTM^*| \end{cases} \quad \text{with}$$

$$|avTARGET - avT^*| = \min_{k-1} |avTARGET - avT_k| \quad \text{and}$$

$$|avTARGET - avTM^*| = \min_{k} |avTARGET - avTM_k|$$

In other words, from the total set of average job type runtimes and their pair-wise mean values the single value $RLTV'$ is selected which is closest to the previous calculated ideal value $avTARGET$. Then the individual job J_i is selected and delivered which minimizes $\left|RLTV - avT_{jT(i)}\right|$. For clarity, the whole strategy is explained by an example now. We assume jobs of four types/users $JT_1, \ldots, JT_4$ being in the data base. We also assume the following average runtimes (in minutes):

$$avT_1 = 10, \qquad avT_2 = 60, \qquad avT_3 = 180, \qquad avT_4 = 200$$

Furthermore, the following monitoring values of an asking node are assumed to be estimated so far:

$$avU = 240, \qquad acU = 220, \qquad B = 2, \qquad R = 0$$

This results in $avTARGET = (240 - 220) \cdot 2 = 40$. The average job runtime closest to this value is $avT_2 = 60$, the closest mean value is $avTM_1 = \frac{1}{2}(avT_1 + avT_2) = 35$. So we get $RLTV' = 35$. Because of $RLTV = RLTV' + \mathrm{rnd}(-2,2)$, the node gets a job of the (randomly chosen) type JT_1 or JT_2. Fig. 2 explains the example.

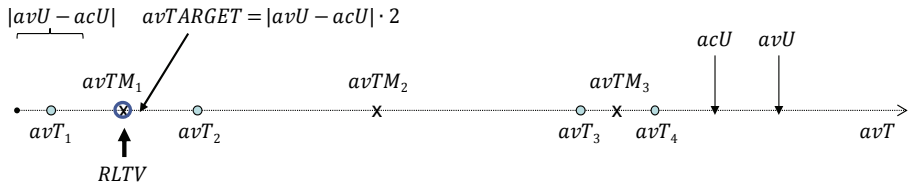


Fig. 2 Uptime-based distribution example

Due to the usage of the mean values and the randomization, the system performs a stochastic selection between the two best suited job types, if the ideal target value $avTARGET$ resides in the mean of two neighbored job types. The example shows also, why the largest average runtime (avT_m, in the example avT_4) is excluded when calculating $RLTV$. Otherwise, the nodes with the largest remaining uptime would work for the user with the longest running jobs only. In other words, a node gets

- a job of type JT_k if $avTARGET$ lies in the interval I_k or
- a job of type JT_k or JT_{k+1} (selected randomly), if $avTARGET$ lies in the interval $I_{k,k+1}$.

The interval bounds for the example above are graphically shown in Fig. 3. In general, the intervals are described as follows (m types, $k \in \{1, \dots, m-1\}$):

- $I_k = \left[\frac{1}{2}(avTM_{k-1} + avT_k), \frac{1}{2}(avT_k + avTM_k)\right]$, where
  $I_1 = \left[0, \frac{1}{2}(avT_1 + avTM_1)\right]$
- $I_{k,k+1} = \left[\frac{1}{2}(avT_k + avTM_k), \frac{1}{2}(avTM_k + avT_{k+1})\right]$, where
  $I_{m-1,m} = \left[\frac{1}{2}(avT_{m-1} + avTM_{m-1}), \infty\right[$

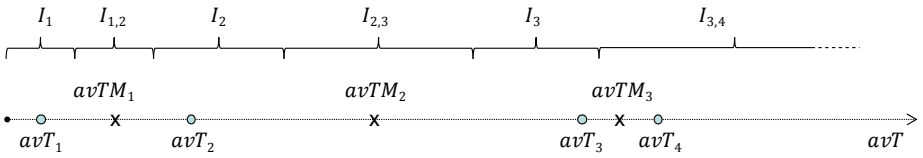


Fig. 3 Interval bounds example

2.4 Simulation

To test the strategies, a simulator was developed. It allows seeing how the system would behave in a specific job and node situation. Every simulated node has to be parameterized in XML:

- $power$: The bench value rB which the node estimates in the beginning (see 2.2) and sends to the server.
- $zerofp1$, $zerofp2$: The number of simulation steps the node is capable to work without failure.
- $incfp1$, $incfp2$: Within this time span the failure probability increases linearly (after the period of working reliable).
- $fail1$, $fail2$: The maximum probability (which is finally reached after the increasing period) for this node to fail during every simulation step.
- cnt: The number of nodes which are parameterized in this manner.

First, all nodes are simulated using the parameters $power$, $zerofp1$, $incfp1$ and $fail1$. After the first third of the simulation is done, all nodes switch to the second parameter set ($power$, $zerofp2$, $incfp2$ and $fail2$). Hence, it is possible to check the behavior of the scheduling algorithm in case of a change in the reliability characteristics of a node. For clarity, the simulated characteristic is explained by an example:

```
<clients>
   <client cnt="5" power="4000"
           zerofp1="60" incfp1="40" fail1="3"
           zerofp2="20" incfp2="90" fail2="1"/>
   …
</clients>
```

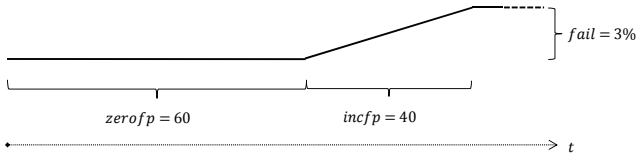


Fig. 4 Reliability behavior example

The five such configured nodes would perform as if they had done the initial benchmark in 4000 milliseconds. After startup, they would work 60 steps without failure and then, within a period of 40 steps, fail with increasing probability per step, up to a maximum value of 3% per step. After failing, the period would start again from the beginning with 60 new reliable steps. After the first third of the total simulation, the nodes would switch to the second parameter set (20, 90, 1).

Every simulated job type ($jobtype$) can be parameterized by the number of its jobs (cnt), the average duration ($jobduration$) of a job belonging to this type (if running on a node with $power = 5000$) and the number of simulation steps ($steps$) the simulator has to execute after adding these jobs:

```
<simulation>
   <step cnt="4000" jobtype="job_10" jobduration="10" steps="0"/>
   <step cnt="800" jobtype="job_50" jobduration="50" steps="0"/>
   <step cnt="200" jobtype="job_200" jobduration="200" steps="0"/>
   <step cnt="100" jobtype="job_400" jobduration="400" steps="5990"/>
</simulation>
```

Out of the many simulated configurations, the one shown above indicates perfectly (Fig. 5, Fig. 6), what one could achieve by using intelligent distribution strategies: Even in difficult situations (in practice, the real future behavior of the nodes and the real future runtimes of the jobs are fully unknown in principle!) the so called *makespan* is improved. The makespan describes the overall time needed to process all available jobs. We simulated a very heterogeneous set of 120 nodes (to save space, their XML specification is omitted here).

The following diagrams show for each job type the percentage of jobs which are done at a specific simulation step. Every type produces a total workload of $jobduration \cdot cnt = 40000$ steps, so in an ideal distribution case all types should reach the 100%-line at the same time. This would state that every user gets the same amount of (successfully used) computing power.

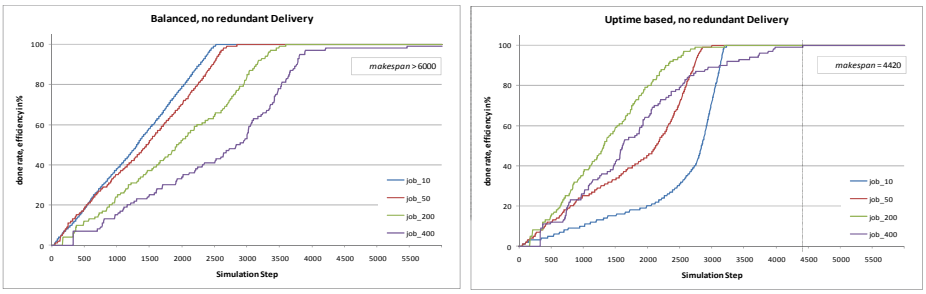


Fig. 5 Balanced and uptime-based distribution, no redundancy

Fig. 5 compares the balanced and the uptime-based distribution. Using the uptime-based strategy, the makespan improves significantly. It also explains how the uptime-distribution achieves this improvement: the short jobs are used to fill the remaining uptimes until the expected failure of the nodes.

But the diagrams also show a problem which typically occurs, when the system has to distribute long running jobs only. Then, they are processed by the not well-suited nodes, too. As the number of available jobs drops further, it could happen that reliable nodes will not get a job because all jobs are already running (maybe on the unreliable nodes). This is caused by the fact the server always delivers a job, independent on the polling node's quality. To guarantee the execution of the jobs by the reliable nodes, an additional, strategy-independent, redundant job execution was implemented: When an agent asks for a job and no free jobs are available, the server delivers one which is already running.

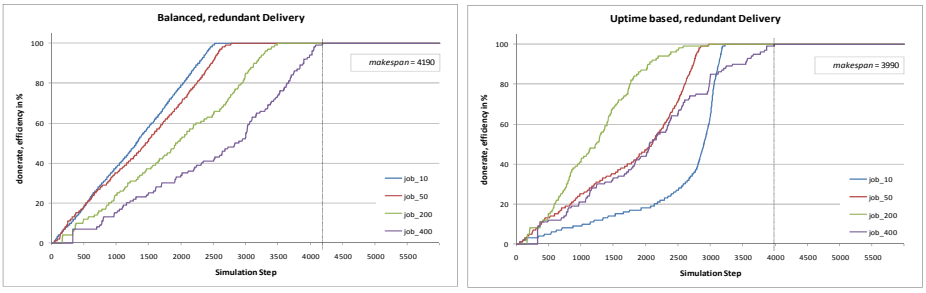


Fig. 6 Balanced and uptime-based distribution with redundancy

Fig. 6 shows the improvement achieved by the redundant delivery. The balanced strategy gains a large makespan improvement. The uptime-based mode improves too, but less significantly, because it finishes the long running jobs earlier, even without redundant delivery. So the negative effect mentioned above does not occur as strongly.

In practical scenarios, a mixture of the balanced and the uptime-based strategies, controlled by a parameter $fairlevel \in [0,1]$, is used:

(1) As necessary for the balanced strategy, for each job type JT_k, $nrWORKING_k \in \mathbb{N}$ (the number of running jobs of JT_k) is estimated.

(2) If $\min_k nrWORKING_k / \max_k nrWORKING_k < fairlevel$, the balanced strategy is used.

(3) Otherwise, the scheduler selects the best uptime-based job.

So the degree of balance/fairness is configurable and normally set to 0.1–0.3.

3 Agent and Usage

Initially, this section describes the autonomous component of the system, the agent which runs on the worker nodes, and then gives a short overview of the graphical tool needed by the user to manage his jobs.

3.1 Agent

The agent is the component that is running on the worker nodes. It requires no user-interaction and behaves fully autonomous, so it can be started as a background service, optionally with an integrated auto-updating mechanism, and performs the following tasks:

(1) Initially, it has to determine the local system parameters that are essential for job selection and execution. These are the installed memory, the CPU architecture (x86/x64), the availability of some runtime environments (.NET, Mono, Java, Python, Perl, GAMS), and the operating system (Windows- or Unix-like).

(2) It starts the working loop with a query to the server for a job. If the server responds with a job description, the agent downloads the specified files.

(3) The agent starts the job with low operating system process priority and waits for its completion. During the execution, it periodically contacts the server to submit the heartbeat. Standard and error out are redirected, if desired by the user.

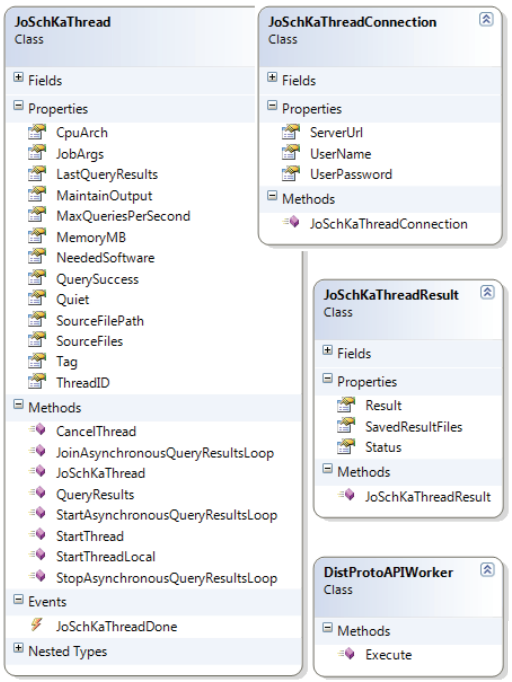
(4) When the job has finished with no error, all result files and the standard/error out data are uploaded to the server.

(5) If all files are transferred successfully, a final commit is sent to the server. Then, and only then, the job is accepted as successfully done and the agent proceeds by sending the next job query to the server (step (2)).

3.2 Parallelization and management tool

It is very easy for a developer to distribute concurrent jobs using JoSchKa. When using the batch system, the application has to be split in single pieces; each of them must (formally) be described by:

- a set of input files $I = \{i_1, \dots, i_n\}$,
- a command (or script) C to be executed on the worker node and
- a set of output files $O = \{o_1, \dots, o_m\}$ which are produced by the command or the script respectively.



If a problem is decomposable to such units $J = (I, C, O)$, and C is executable on the worker nodes, it can be distributed and run as batch jobs by the described system.

A user, who wants to create a more interactive parallel application, create jobs programmatically and react directly on their results, can use the JoSchKa thread API for .NET to create a parallel program. With this API it is possible to dynamically upload program code to the server, start threads executing this code and query their results. So, developing a distributed program is similar to the development of a multi-threaded local application.

Fig. 7 The JoSchKa API classes

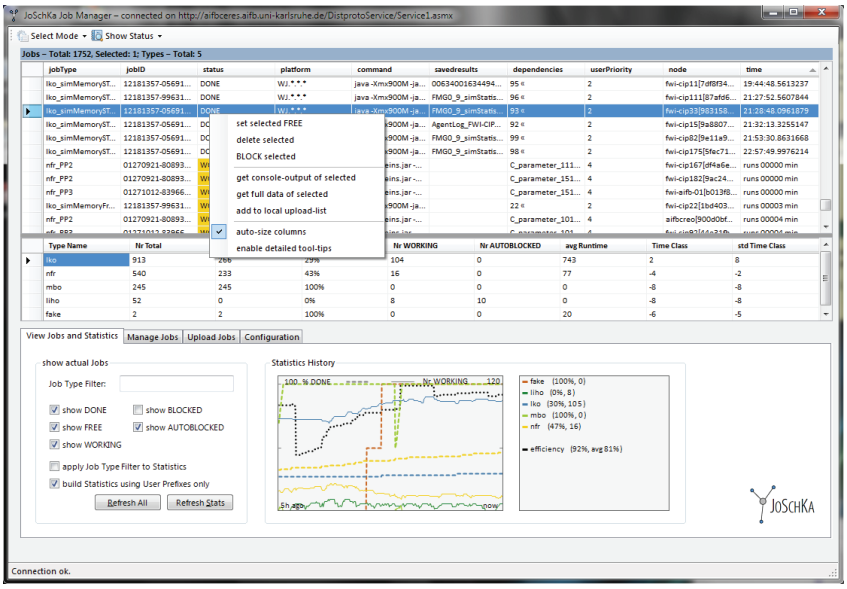


Fig. 8 Job management tool

Due to the lack of space, more details about the API and the graphical management tool are omitted, we just give a simplified API class diagram (Fig. 7) and a screenshot of the frontend showing the state of the submitted jobs, their execution status and a context menu providing some management functions (Fig. 8). For more information, please refer to [5].

4 Relation to Organic Computing

Organic Computing (OC) [10] focuses on self-organizing systems adapting robustly to dynamically changing environments without losing control. It proposes a generic observer/controller architecture, which allows self-organization but enables reactions to control the overall behavior to the (technical) system [11]. We have both a system under observation and control (SuOC), and observing and controlling components, which are available for monitoring and influencing the system. The observer has to measure, quantify and predict further behavior of the monitored entities. The aggregated values are reported to the controller, which executes appropriate actions to influence the system. The overall goal is to meet the user's requirements. The user himself can influence the system by manual changing the objective function or by directly accessing the observed/controlled entities (Fig. 9).

The described situation of computational jobs, heterogeneous unreliable job executing worker nodes and a distribution system like JoSchKa is a good example for such an organic system. The user-given job data and the worker nodes represent the system under observation and control. In an autonomous way, they execute the jobs, while failing and rebooting in an uncontrollable way.

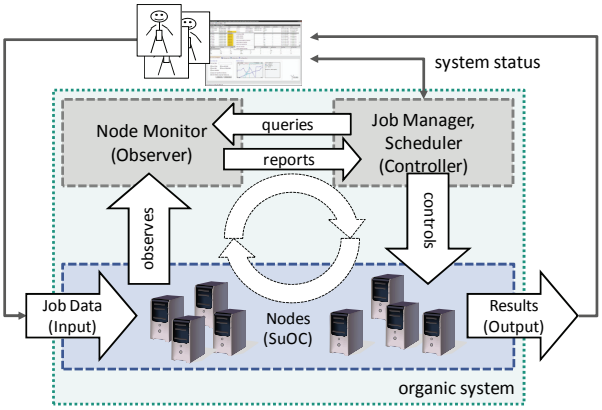


Fig. 9 The generic O/C architecture

The monitoring component of the JoSchKa system has the role of an observer while the job selection component (scheduler) behaves as an OC controller. It does not control the nodes directly, of course, but with its intelligent job assignment (based on the observer's measured values and predicted node-behavior) it helps achieving all users' goals: The completion of all jobs (i. e. the production of result files) as fast and early as possible. The users can influence the system by manually blocking or deleting their jobs using the management GUI; but normally they do not have to, because the nodes and the distribution system act fully autonomous.

5 Practical Experiences and Outlook

Since 2006, JoSchKa runs on PCs of a computer pool of the Faculty of Economics and Business Engineering of the Karlsruhe Institute of Technology (KIT) and on some other small pools, completed by a few servers. The pool PCs are standard Windows or Linux desktops, equipped with 2–3 GHz CPUs and 1–2 GB of memory. Since then, the nodes finished about 250 000 jobs, mainly belonging to naturally inspired optimization algorithms and organic computing simulations [12]. The students working interactively on the desktop did not notice the permanent load. As showed in [5], the execution time of standard desktop applications does not slow down when running a CPU-intensive background task with lowest process priority.

The students and the scientific staff of our faculty benefit well by JoSchKa: Students normally develop their optimizing programs in Java on a Windows platform, as they learned it during their education. But the Unix-based HPC or Grid platforms of the computing centre require C/C++ or – even worse – Fortran, so the usage of them is quite difficult, especially at the end of a students' thesis work, when time runs out. The convenient handling of JoSchKa gave them the possibility to run the simulations concurrently on multiple worker nodes, even when a distributed execution was initially not intended.

Due to the raising availability and usage of cloud computing [13] providers (Amazon EC2, Google AppEngine or Microsoft Azure, e. g.), it's imaginable to extend JoSchKa for cloud usage in the near future. If a user needs unsatisfiable computing resources, it should be possible to dynamically create them by using a virtual infrastructure as provided by a cloud service. The system tries to estimate how many new machines are necessary and how they have to be equipped (hard- and software), and

tells the user the price for these virtual machines. If he agrees, the virtual machines are cloned from templates; configured according the user's needs, and started to execute the agents. So JoSchKa would not only react on the dynamic behavior of fixed machines, but also would extend/reduce them dynamically (and thus acting as a real intervening organic controller), if needed.

References

[1] Anderson, D. (2004). BOINC: A System for Public Resource Computing and Storage. 5th IEEE/ACM International Workshop on Grid Computing, (pp. 365–372). Pittsburg, PA.

[2] Wedeniwski, S. (2008). ZetaGrid – Computations connected with the Verification of the Riemann Hypothesis, Foundations of Computational Mathematics Conference, Minnesota, USA, August 2002.

[3] Litzkow, M., Livny, M., & Mutka, M. (1988). Condor – a Hunter of idle Workstations. 8th International Conference on Distributed Computing Systems, (pp. 104–111).

[4] Luther, A., Buyya, R., Ranjan, R., & Venugopal, S. (2005). Peer-to-Peer Grid Computing and a .NET-based Alchemi Framework. In L. Y. Guo, High Performance Computing: Paradigm and Infrastructure. New Jersey, USA: Wiley Press.

[5] Bonn, M. (2008). JoSchKa: Jobverteilung in heterogenen und unzuverlässigen Umgebungen. Dissertation an der Universität Karlsruhe (TH), Universitätsverlag Karlsruhe, 2008.

[6] Gelernter, D. (1985). Generative Communication in Linda. ACM Transactions on Programming Languages and Systems , 7 (1), pp. 80–112.

[7] Dostal, W., Jeckle, M., Melzer, I., & Zengler, B. (2005). Service-orientierte Architekturen mit Web Services. Spektrum Akademischer Verlag.

[8] Tsafrir, D., Etison, Y., & Feitelson, D. (2007). Backfilling using System-generated Predictions rather than User Runtime Estimates. IEEE Transactions on Parallel and Distributed Systems , 18 (6), pp. 789–803.

[9] He, Y., Hsu, W., & Leiserson, C. E. (2007). Provably efficient two-level adaptive Scheduling. In U. Schwiegelshohn (Hrsg.), 12th International Workshop JSSPP 2006. LNCS 4376, S. 1–32. Saint-Malo, France: Springer Verlag Berlin Heidelberg 2007.

[10] DFG Priority Program 1183 Organic Computing. http://www.organic-computing.de/SPP, 2005. Visited Sept. 2009.

[11] Richter, U., Mnif, M., Branke, J., Müller-Schloer, C., & Schmeck, H. (2006). Towards a generic observer/controller architecture for Organic Computing. In Christian Hochberger and Rüdiger Liskowsky, INFORMATIK 2006 – Informatik für Menschen!, volume P-93 of GI-Edition – Lecture Notes in Informatics (LNI), pp. 112–119. Bonner Köllen Verlag, 2006.

[12] Richter, U., & Mnif, M. (2008). Learning to control the emergent Behaviour of a multi-agent System. Proceedings of the 2008 Workshop on Adaptive Learning Agents and Multi-Agent Systems at AAMAS 2008.

[13] Baun, C., Kunze, M., Nimis, J., Tai, S. (2009). Cloud Computing: Web-basierte dynamische IT-Services. Springer, Berlin; Auflage: 1.