Proceedings of the International Multiconference on Computer Science and Information Technology pp. 327–335
ISSN 1896-7094

Efficient Matchmaking in an Agent-based Grid Resource Brokering System

Mateusz Dominiak[1], Wojciech Kuranowski[2], Maciej Gawinecki[3], Maria Ganzha[3,4], and Marcin Paprzycki[3,5]

[1] Faculty of Mathematics and Information Science, Warsaw University of Technology
[2] Wirtualna Polska, Software Development Department
[3] Systems Research Institute, Polish Academy of Science
[4] Department of Administration, Elbląg University of Humanities and Economy
[5] Computer Science Institute, SWPS

Abstract. Centralized *yellow pages*-based approach is one of well-known methods of facilitating resource discovery in distributed systems. Due to its conceptual simplicity, it became our choice in two recent research projects: agent-based model e-commerce system and agent system for resource brokering in a grid. It is the latter project that is used as a background to study efficient ways of implementing agent-based *yellow page* service. Results of actual experiments provide us with guidelines how to maximize its throughput.

1 Introduction

In [1] we have presented a conceptual framework for an *agent team*-based approach to resource brokering in a grid. This work follows and expands results presented in [7, 8]. In all cases grid is conceived as a dynamic environment, in which *worker agents* contribute their resources and are remunerated for their usage, while *user agents* locate resources to execute jobs supplied by their owners. Therefore, we deal with a standard situation where matchmaking (between resources and users) has to take place. In [9], authors have discussed four approaches to matchmaking in distributed systems. Since each one of them has its advantages and disadvantages, in our earlier work (see [3] and references collected there) we have decided to utilize a "yellow page" based approach. In our present work matchmaking is also facilitated by the *Client Information Center* (*CIC*) service—implemented as a *CIC agent infrastructure*. The aim of this paper is to discuss the way in which the *CIC* service can be efficiently implemented to prevent it from becoming a bottleneck of the system.

We proceed as follows, in the next section we summarize the design of our system. We follow with the description of three basic *CIC* agent infrastructures and their experimental performance. In Section 4 we introduce an improved version of the best of the three infrastructures and report its performance. There we also discuss a number of auxiliary topics related to the way in which the *yellow page* service is to be implemented.

2 Agent-team-based grid resource brokering

Let us start by briefly summarizing main ideas behind the proposed *agent team*-based grid resource brokering system (its complete description can be found in [1]). Our most basic assumption is that a **single** *worker agent*, for example representing a typical "home-user", has limited value in real-life open-grid applications. While we recognize success of applications like *SETI@home* that is based on harnessing power of millions of "home-PC's," this application (and a number of similar ones) has very specific nature. There, the fact that a particular resource "disappears" during calculations is rather inconsequential, as any data item can be processed at any time and in any order. Furthermore, data item that was not completed due to the "vanishing PC" can be completed in the future by another resource. This, however, is not the case in realistic (e.g. business-type) applications, where calculations have to be completed in a specific order and, usually, within a well-defined time-frame. In other words, in most applications some form of a service-level agreement (*SLA*), that assures conditions of job completion has to be involved. Assuring such *SLA* in the case of a "home-PC" that, for instance, does not have a UPS (which is typically the case), can be rather difficult. Therefore, to address this problem, we introduce virtual organizations, called *agent teams*. Each team consists of a number of *worker agents* and a leader, the *LMaster agent*. It is the *LMaster* with whom *user agents* negotiate terms of task execution, and who decides whether to accept a new *worker agent* to the team.

For an agent team to be visible to potential users or team members, it must post its *team advertisement* in an easily reachable way. As described in [9], there are many ways in which information used in matchmaking can be posted in a distributed system and each one of them has advantages and disadvantages. In our work we have decided to utilize a *yellow page*-type approach and thus *LMaster* agents post their team advertisements within the *Client Information Center* (*CIC*). Such advertisements contain both information about offered resources (e.g. hardware capabilities, available software, price etc.) and "team metadata" (e.g. terms of joining, provisioning, specialization etc.). In this way *yellow pages* can be used: (1) by *user agents* looking for resources satisfying their task requirements, and (2) by *worker agents* searching for a team to join. For example, *worker agent* representing a computer with installed Maple software, may want to join a team specializing in solving problems utilizing Maple. Note that an *agent team* may assure the *SLA*, as in the case when one PC goes down it will be able to immediately run the job on another and complete it in time or almost in time.

In our system, the user initiates the execution of the job by providing its *user agent* with specific requirements such as: *resource requirements*—for completing the task, and *execution constraints*—time, budget etc. From this moment on, the *user agent* autonomously acts on behalf of its owner. First, it queries the *CIC* for resources matching specified requirements. In response it obtains a list of query-matching teams. Then it negotiates with their *LMasters*, taking into account its *execution constraints*, to find the best team for the job.

Similarly, the user can specify that its agent joins a team, and provide it with conditions for joining (e.g. frequency of guaranteed jobs or share of generated revenue). In this case the *user agent* queries the *CIC* and obtains list of teams of possible interest; negotiates with them, decides which team to join and starts working for it. Observe that in all situation involving agents initiating interactions with the system, they have to interact with the *CIC* first. Note also that, since our system follows the general tenets of agent-system design, the *CIC* service has to be designed and implemented as a *CIC agent* (possibly supported by auxiliary agents). This being the case, the question arises: how can we make this agent-infrastructure as efficient as possible? To find an answer, we introduce a number of possible architectures of the *CIC agent* and its co-workers and empirically establish their performance.

3 CIC architecture and implementation

As indicated above, the *CIC* infrastructure is one of key components of our system. Therefore, it must be capable of efficiently handling large number of requests. Specifically, since interactions between *user agents* and the *CIC* are the key part of early stages of job execution, or *user agent* joining an agent team, long delays in *CIC* responses would become a major bottleneck of the system.

In this context, let us note first, that in our system the *yellow page* information is stored in an ontologically demarcated form [1]. To facilitate it we use Jena 2.3 [5] and its database persistency mechanism (see [6] for a report on using Jena with massive store of ontological triples). Second, the main goal of our work is a very practical and follows our earlier studies in efficiency of our agent platform of choice—JADE [4]. In [2] we have shown, among others, that tasks involving database access can be efficiently distributed to multiple database-access agents (*SQLAgents*). Specifically, in the reported experiment, a single agent was receiving and enqueuing client-requests, and multiple *SQLAgents* were dequeuing requests and executing them on the database. All query-processing agents and the database were running on separate computers. Multiple tests with different number of *SQLAgents* have been executed and have shown that as the number of *SQLAgents* increases to 5, the total query-processing time decreases by almost 33%. Obviously, we could try utilizing such agent-based database access mechanism in our system. More generally, we have decided to look for the most efficient agent-based architecture for the *CIC* service and as the first step implemented the following three basic architectures:

1. multi-threaded CIC, see figure 1.
2. multi-agent CIC with local database agents (*CICDbAgents*), see figure 2.
3. multi-agent CIC with distributed database agents—located on separate computers (based on the idea from [2]), see figure 3

In (1) we utilize the well-known task-per-thread paradigm. We have used Java threads and made them accessible to the *CIC agent* within its container. Each worker thread has its own connection to the database and its instance of the Jena model. Initialization of these resources is computationally expensive and that is why instead of spawning new threads, we use preinitialized threads in the worker thread pool. The *CIC* agent picks requests (query-requests or yellow-pages-update-requests) from the JADE-provided *message queue* (storing incoming standard ACL messages) and enqueues them into the

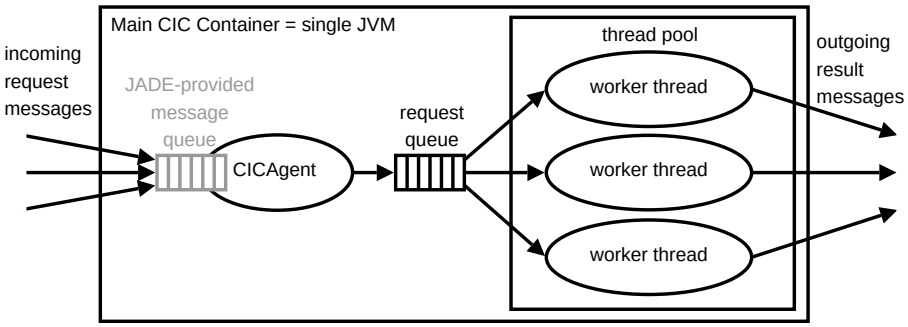

Fig. 1. Request/result flow in a multi-threaded *CIC* architecture.

request queue (which we have implemented). It is this *request queue* from which free worker threads pull requests for execution. After executing the query they send obtained responses to their originators.

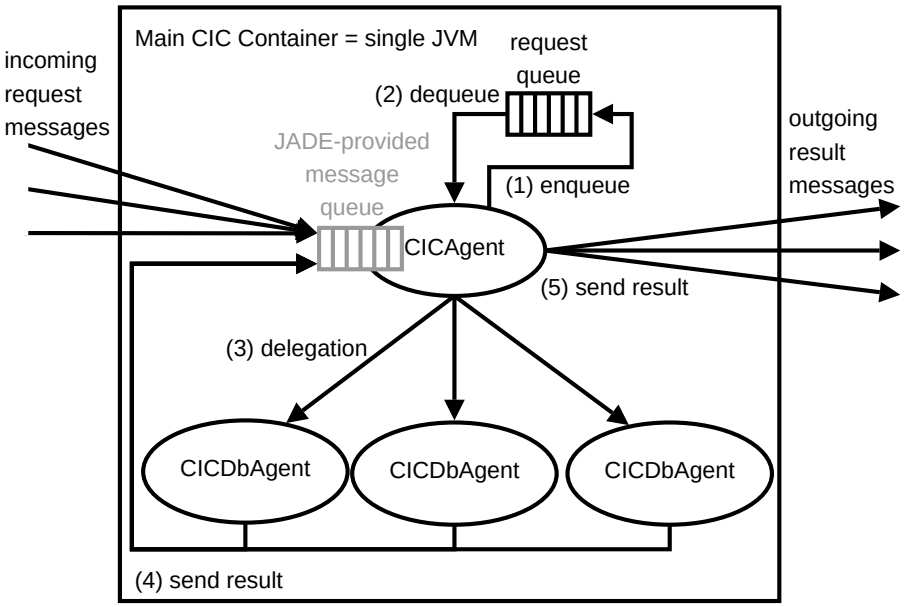

Fig. 2. Request/result flow in multi-agent *CIC* architecture with local *CICDbAgents*.

In the second approach we use local (residing in the same agent container) *CICDbAgents*—instead of worker threads. The *CIC* agent picks requests from the JADE-provided *message queue* and enqueues them into, implemented by us, internal *request queue*. This queue acts as a buffer between the *CIC agent* and the *CICDbAgents* and, furthermore, reduces the number of messages stored in the JADE *message queue*. Note that this queue is the only way for the *CIC* to receive ACL messages. Incoming requests are **delegated** (in the form of ACL messages) to "free" *CICDbAgents* by the *CICAgent*. Each database agent completes one task (request) at a time. Upon completion, results are sent (also as an ACL message) back to the *CICAgent*. As a result they are placed in the same JADE *message queue* as the incoming query-requests. There are two behaviors within the *CICAgent* that are servicing the JADE *message queue*. One of them checks for incoming query-requests, while the other checks for incoming query-results. Since both behaviors operate within a single thread (JADE utilizes a one-thread-per-agent paradigm), it can be assumed that (except when there is nothing to do for one of them) they take turns removing messages of a given type from the JADE *message queue*. As we will see in the next section, this has very important consequences for the performance of this approach.

The last approach (3), is almost exactly the same as the previous one (2). The only difference is that database agents are located on remote machines contributing additional computational power and allowing *CICDbAgents* to work without stealing resources from the *CICAgent*.

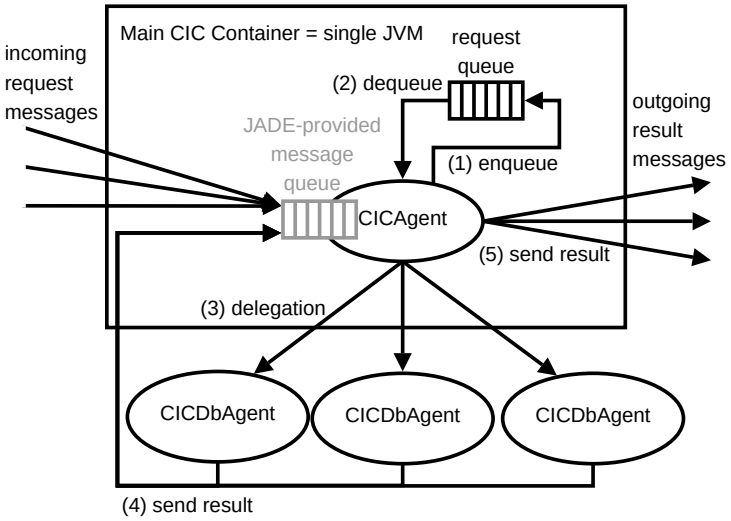

Fig. 3. Messages flow in multi-agent *CIC* architecture with distributed *CICDbAgents*.

Overall, in the multi-threaded approach (1) we utilize a "pull architecture", where worker threads pick requests from *request queue*, while in multi-agent solutions (2,3) requests are delegated by the *CICAgent* to *CICDbAgents*—"push architecture".

3.1 Experiments with the three basic architectures

In our experiments, to simulate a flow of incoming requests from *user agents* we used 4 *Querying Agents* (QA), requesting the *CIC* to perform SPARQL [10] resource queries. It should be noted that the form of the SPARQL query can change performance of the system. The ARQ engine used in Jena, is responsible for executing the query on RDF resources persisted in the database. It translates only parts of the SPARQL query into SQL. The remaining parts (e.g. FILTER operations) of the SPARQL query are not performed through the SQL query, but locally by the ARQ engine, utilizing local JVM resources. In our case queries had the following form:

```
PREFIX grid: <http://gridagents.sourceforge.net/Grid#>
SELECT ?master
WHERE {
    ?comp grid:cpuClockSpeedMhz ?cpu .
    FILTER (?cpu > "1000"^xsd:integer) .
    ?master grid:offersResource ?comp .
}
```

Each *QA* was running concurrently on a separate machine, and was sending 2,500 requests and receiving query-results. Thus in each experiment 10,000 queries have been processed by the *CIC*. Since we have been running multiple experiments (especially when attempting at performance tuning), we have developed an experimental framework for running tests automatically, while varying their parameters (e.g. number of worker threads, number of *CICDbAgents* etc.). All experimental runs were coordinated by the *Test Coordinator Agent* (*TCA*). Before each test, remote JADE agent containers were restarted to provide equal environment conditions. Experiments were performed using up to 11 Athlon 2500+, 512MB RAM machines running Gentoo Linux and JVM 1.4.2. Computers were interconnected with a 100Mbit LAN. The MySQL 4.1.13 database used by Jena persistence mechanism for storing *yellow pages* data was installed on a separate machine. In all cases the experimental procedure was as follows:

1. Restart of remote agent containers
2. Experiment participants send *ready* message to the *TCA*—just after they are set-up and ready for their tasks
3. On receiving the *ready* signal from *all* agents, the *TCA* sends *start* message to all *QA*s, triggering start of the experiment
4. When *QA*s receive all results back, they send a *finish* message to the coordinator (the *TCA*)
5. Reception of all *finish* signals ends the experiment

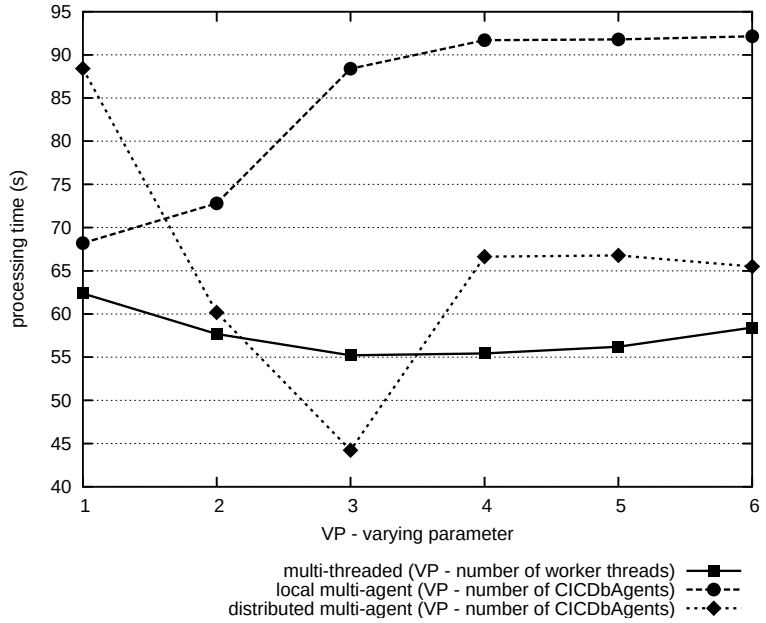

Fig. 4. Experimental results of different *CIC* architectures: multi-threaded, multi-agent with local *CICDbAgents* and multi-agent with distributed *CICDbAgents*; processing time of 10,000 queries for varying number of worker threads in multi-threaded architecture/number of *CICDbAgents* in multi-agent architectures.

In figure 4 we represent the total processing time of 10,000 requests by each *CIC* architecture, when the number of agents/worker threads increases from 1 to 6. Obtained results are as expected for the first approach (worker-threads): as the number of threads increases from 1 to 3 we can see a total time reduction of about 11%. Further increase of the number of threads does not result in performance increase indicating, that all local resources have been consumed when 3 threads were used.

What is somewhat surprising is the fact that the architecture with local *CICDbAgents* does not scale at all. In order to understand this situation we have to refer to the mechanics of messaging in JADE, which provides each agent with its own *message queue* managed by the JADE Message Transport System (MTS). In this context let us recall that as the system works, query-request messages are intermixed with response messages. At the same time, we have two message retrieving behaviors that take turns trying to retrieve their types of messages from the queue. Let us now observe what happens in early stages of our experiment, when the response-message retrieving behavior tries to retrieve a response-message. This process involves iterative filtering through messages stored in the *message queue*. Thus, if there are 1,000 request messages stored in the queue when the first response message was received from the *CICDbAgent*, it will be placed at the end of the queue. In order to get to that message, all 1,000 request messages have to be checked first, and finally the result message will be found at the end of the queue. Obviously, as time passes, and request messages become intermixed with result messages, the situation becomes less radical, but still this approach turns out to be relatively inefficient. Further support for our explanation can be found in figure 8 (even though this figure represents performance of the last approach, behavior noted in the case of local agents was very similar). There we can observe that, as the number of request messages decreases, throughput increases. Obviously, this effect is somewhat biased by the way in which our experiment was set-up; all query-request messages were send at once at the beginning and then *Querying Agents* were just waiting for responses. Note also that this situation cannot be changed programmatically, since this is how JADE works internally.

As it turns out, the best performance can be observed in the case of the multi-agent approach with 3 distributed *CICDbAgents*. In this case, reduction of time of order 2.5 is observed. Since the starting point is well above the case of the local architecture (caused by the cost of computer-to-computer communication), the maximum reduction of time against the threaded solution is of order 18%. Unfortunately, as the number of agents becomes larger than 3, the same effect as in the case of local agents can be observed—performance decreases due to the way in which the *CICAgent* removes data from the message queue.

Overall, as the result of our initial set of tests, we were able to establish importance of the way that the messaging is handled. Specifically we have found that (a) when only a single machine is available for facilitating the *yellow page* service, a threaded solution should be used, and (b) the additional com-

putational power available in the case when *CICDbAgents* are located on multiple separate machines plays an important role and makes this particular approach the best candidate for further performance improvement.

4 Performance tuning and auxiliary topics

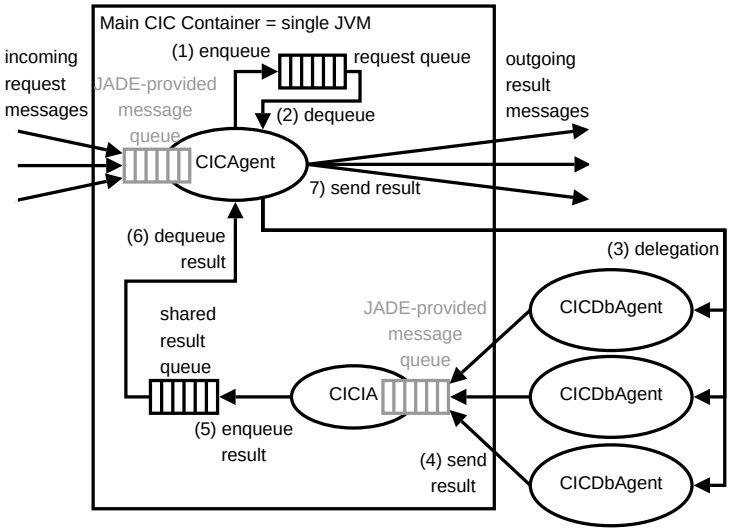

Fig. 5. Request/result flow in distributed multi-agent *CIC* with *CICIA*.

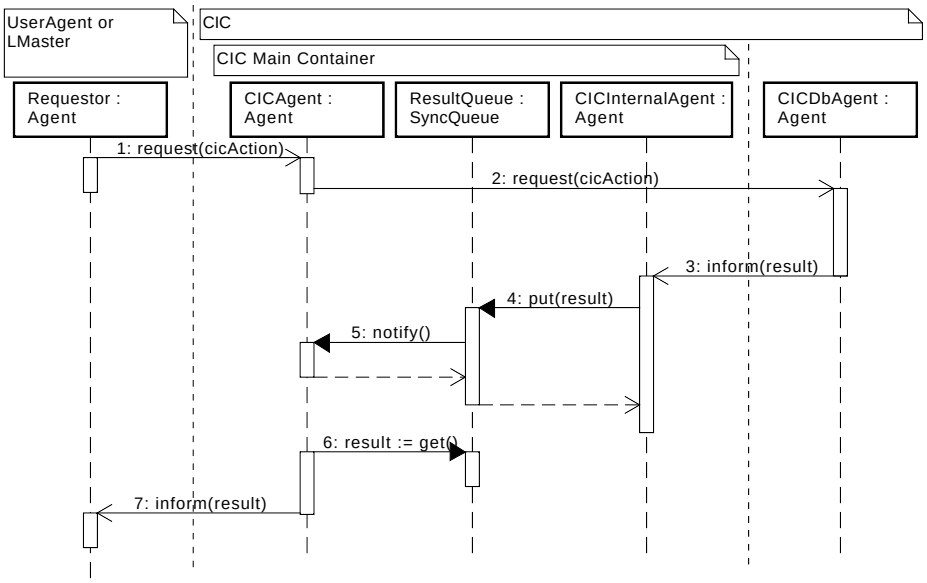

Fig. 6. Sequence diagram: Handling requests by *CIC* with additional *CIC Internal Agent*.

Based on results collected thus far we have decided to attempt at improving the performance of the third *CIC* architecture—the multiagent approach with agents located on separate machines. To overcome the way that query and response messages are handled, we have added the *CIC Internal Agent* (*CICIA*) (see figure 5). This agent plays a role of an intermediary between the *CICAgent* and *CICDbAgents*. More

precisely, processing request messages starts in the same way as in the previous solution—these messages are stored in JADE-provided *message queue*. The *CICAgent* removes messages from the *message queue* and stores them in the internal *request queue* and, later, delegates them to the *CICDbAgents* (sending them as ACL messages). *CICDbAgents* query the database and send results of their queries to the *CICIA* (as ACL messages—to be stored in the JADE *message queue* of the *CICIA*). Upon reception of such messages, *CICIA* enqueues them into a synchronized *result-queue*, from which they are dequeued by the *CICAgent* and send back to requesters. In other words, the intercommunication between the *CICIA* and the *CICAgent* is accomplished through a shared *result-queue* instead of ACL messaging. As it is easy to see, this is also why these agents (*CICIA* and *CICAgent*) must run within the same agent container (the *Main CIC Container)*. The *CICAgent* has now three behaviors: (1) receive request-message from the JADE message queue and enqueue it in the request-queue, (2) dequeue request from the request-queue and send it to the *CICDbAgent*, and (3) dequeue message from the result-queue and send it to the requester. The sequence diagram of handling the request is presented in figure 6. Observe that, in the modified approach, database query results are **not intermixed** with query requests and hence we eliminate the above mentioned overhead of filtering results form the JADE message queue.

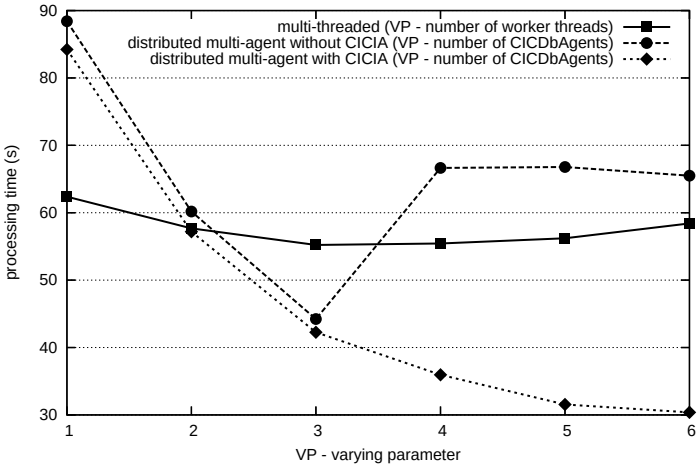

Fig. 7. Comparison of multi-threaded *CIC* with distributed multi-agent architecture with and without *CIC Internal Agent*; processing time of 10,000 queries varying number of worker threads/number of *CICDbAgents*.

In Figure 7 we compare performance of the *CIC* service implemented using worker threads and two versions utilizing non-local *CICDbAgents*—with and without the *CICIA*.

As can be seen, the performance of both non-local *CICDbAgent*-based approaches, when the total number of *CICDbAgents* is between 1 and 3 is quite similar (the architecture with the *CICIA* is slightly better). However, as the number of *CICDbAgents* increases up to 6, the performance improves steadily. We can also observe a leveling-off effect. Therefore, the fact that we were not able to run experiments with more than 11 computers (and thus the largest number of *CICDbAgents* was 6), seems rather inconsequential. Specifically, it can be predicted that if the number of *CICDbAgents* was to increase further, then the performance gain would be only marginal. Overall, with 6 *CICDbAgents* the performance gain over the system with only a single *CICDbAgent* is of order of 3. Furthermore, the performance gain over implementation based on threads is of order 2 (here we compare the best threaded solution—with 3 threads—with the distributed agent solution with *CICIA*, for 6 *CICDbAgents*).

Finally, in Figure 8 we present the throughput of the two systems with non-local *CICDbAgents* (with and without the *CICIA*). These results were collected using another *CICAgent* behavior which was controlling the state of the *CICAgent* and logging appropriate variables for post-processing. The results confirm our earlier understanding of processes taking place in the system working under conditions of our experiment. In the case of the system without *CICIA*, throughput is slowly increasing as more and more response messages are intermixed in the queue with request messages. In this way, time to retrieve a response message decreases (these messages move closer and closer to the front of the JADE *message-queue*). When all request messages have been processed (e.g. have been removed from the *CICAgent* JADE message-queue), the request retriever behavior blocks and the response message retriever starts to "continually" retrieve results form the *message queue* and send them to the requesters. This situation

can be observed in the form of the throughput spike near the end of the process. In the case of the architecture with the *CICIA* we observe (after a brief start-up period) a steady performance of the order of 400–500 processed requests per second.

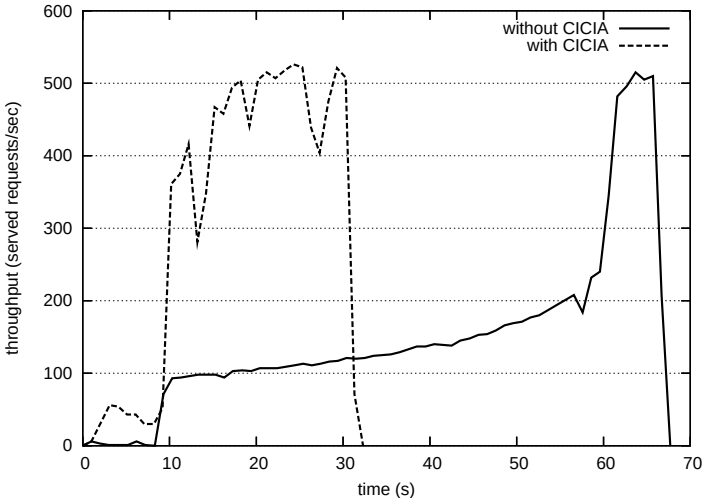

Fig. 8. Comparison of distributed multi-agent architecture (6 *CICDbAgents*) with and without *CIC Internal Agent*. Throughput (served requests per second) vs. time.

It should be mentioned that we have also evaluated a modification of the above architecture, where the *CICIA* becomes both the delegator of requests and receiver of results. This architecture proved to be slightly faster, however it was more complicated from the conceptual point of view. Therefore, for the sake of design readability we have abandoned that idea as the final solution to our problem.

4.1 Additional performance tuning

After establishing candidate architecture, we have proceeded with further performance tuning. In the multi-agent architecture, the *CICDbAgent* is requested to perform a single action, and once it returns result of it there is another request delegated to it. Obviously, proceeding in this way results in *CICDbAgents* being idle for some time—between sending result to the *CICIA* and receiving a new request. We have found that by adding local request queues of size 10 to *CICDbAgents*, the overall performance of the *CIC* increases by approximately 3%.

4.2 CIC reliability

Introduction of additional database agents to the *CIC* architecture reduces its reliability. In the case of failure of the *CICDbAgent*, requests owned by that agent would be lost without any notification to the client. Therefore, we have decided to provide a recovery mechanism. Since all requests and results flow through the *CICAgent* we are able to continually store and maintain two snapshots: (1) one with recent requests delegated to database agents, and (2) one with recently received results. These snapshots have reasonably bounded size that depends on the number of database agents and sizes of their local queues. In the case of a crash of a *CICDbAgent*, recovery procedure finds requests (delegated to that agent) without matching results (in the second snapshot). These requests are put back to the request queue for subsequent execution.

4.3 Introducing prioritized requests

Let us observe that it is extremely important to keep *grid yellow pages* up-to-date as this allows us to limit number of missed query results. Therefore, requests are differentiated according to their type, and any *team advertisement* modification has precedence over querying actions. To facilitate this, we changed implementation of *request queue* into *priority request queue*, which queues requests according to their priority and delegates modification actions for execution first. For example, having $1,000$ pending

query-requests, incoming team modifications will be placed in priority queue before queries, so that the pending queries are "assured" to be served with the most up-to-date information. It should be noted that here we assume that the overall number of modifications of *yellow pages* is negligible with respect to the number of query requests, otherwise starvation of querying actions could occur.

5 Concluding remarks

In this paper we have considered architectural design of the *yellow page* service to be used in an *agent team*-based grid resource brokering system. We have considered three basic architectures and, on the basis of experimental performance analysis, have selected one for further performance tuning. We have shown, that the best solution for *yellow page* service is either: (1) in the case when only a single computer is to run the *yellow page* service—*CICAgent* and a limited number of worker threads, and (2) in the case when at least four computers can be used to run the *yellow page* service—architecture in which at least 3 (in our case) *CICDbAgents* run on separate machines and where an additional *CICIA* is used to handle completed requests. Furthermore, the later architecture can be slightly improved by sending multiple requests send for *CICDbAgents* to process. This is the architecture that we plan to utilize in our system.

References

1. M. Dominiak, W. Kuranowski, M. Gawinecki, M. Ganzha, M. Paprzycki, *Utilizing agent teams in grid resource management—preliminary considerations.* In: Proceedings of the J. V. Atanasov Conference, Sofia, October, 2006 (to appear).
2. K. Chmiel, D. Tomiak, M. Gawinecki, P. Kaczmarek, M. Szymczak, M. Paprzycki, *Testing the Efficiency of JADE Agent Platform,* in: Proceedings of the ISPDC 2004 Conference, IEEE Computer Society Press, Los Alamitos, CA, 2004, 49–57.
3. C. Badica, A. Badita, M. Ganzha, M. Paprzycki, Developing a Model Agent-based E-commerce System. In: Jie Lu et. al. (eds.) *E-Service Intelligence—Methodologies, Technologies and Applications* (to appear)
4. JADE: Java Agent Development Framework. See `http://jade.cselt.it`
5. Jena—A Semantic Web Framework for Java. See `http://jena.sourceforge.net/`
6. K. Portwin, P. Parvatikar, Building and Managing a Massive Triple Store: An Experience Report. See `http://xtech06.usefulinc.com/schedule/paper/18`
7. S. S. Manvi, M. N. Birje, B. Prasad, *An Agent-based Resource Allocation Model for computational grids,* Multiagent and Grid Systems, **1**(1), 2005, 17–27.
8. O. F. Rana, B. Di Martino, *Grid performance and resource management using mobile agents, in: Performance analysis and grid computing,* 2004, 251–263.
9. D. Trastour, C. Bartolini, C. Preist, *Semantic Web Support for the Business-to-Business E-Commerce Lifecycle,* in: Proceedings of the WWW'02: International World Wide Web Conference, ACM Press, New York, USA, 2002, 89–98.
10. SPARQL Query Language for RDF, `http://www.w3.org/TR/rdf-sparql-query/`