WARSAW UNIVERSITY OF TECHNOLOGY
Faculty of Mathematics
and Information Science

Master's Thesis
Computer Science

Intelligent Software Agents in Resource Management on the Grid

Author:
Mateusz Dominiak

Supervisor:
Prof. Marcin Paprzycki

Warsaw, October 2006

....
Author's Supervisor's
signature signature

Intelligent Software Agents in Resource Management on the Grid

Abstract

Grid computing is a promising approach of utilizing geographically distributed, multi-domain computational resources for massive, high performance computations. Performance to cost ratio of grid computing is especially attractive when comparing it with expensive supercomputers. While the grid computing is speeding up recently, the progress is still not satisfactory probably because of overly complicated grid computing infrastructure. It has been recently proposed that software agents might be the solution for creating adaptable middleware for grid computing. The thesis proposes agent-based system for resource management on the grid.

There are *User Agents* that act autonomously on behalf their users striving to execute submitted jobs on the grid utilizing available resources in the way that meets user requirements. Each grid resource has its *Worker Agent* which offers its services to the grid according to the specified resource policy. The system addresses the key challenges of the grid: highly dynamic nature where resources appear and disappear without notice, workload of nodes is susceptible to rapid and unpredictable fluctuations. Therefore *agent teams* are introduced to provide *Quality of Service* and *Service Level Agreements*. *Agent team* are virtual organizations which comprise collaborating *Worker Agents* and *Local Master Agent*, which is the leader of the group.

Agent teams post their team advertisements to the *Client Information Center* (*CIC*) which is centralized grid yellow pages made accessible to all grid clients via *CIC Agent*. Team advertisements, expressed as ontology demarcated data persisted in relational database, consists of list and description of resources that are offered by the team. *CIC* allows *User Agent* to query teams that are capable of executing jobs according to the job requirements given by the user. The interaction with *CIC* occurs in the early stage of almost any system action, hence *CIC* service must be fast, reliable and up-to-date. Three *CIC* infrastructures are implemented: multi-threaded, local multi-agent, distributed multi-agent. Performance results are established empirically and discussed. The best infrastructure is further tuned for more performance boost and its reliability is secured by proper recovery mechanism.

User Agents negotiate with potential teams, obtained from *CIC*, utilizing *FIPA Contract Net Protocol*. The offers proposed by team leaders are validated against marginal constraints specified by the user. The best offer is chosen using *Multi Criteria Analysis*.

Inteligentni Agenci Programowi w Zarządzaniu Zasobami w Gridzie

Streszczenie

Siatka komputerowa (ang. grid) jest obiecującym podejściem do problemu wykorzystania zasobów, które są rozproszone geograficznie i należą do różnych domen administracyjnych, celem ich zaprzęgnięcia do obliczeń wymagających dużej wydajności. Stosunek wydajności do kosztu obliczeń na siatce jest zwłaszcza ciekawy, gdy zestawimy go i porównamy z drogimi superkomputerami. Postęp w dziedzinie obliczeń na siatce przyspiesza, jednak wydaje się on nie być satysfakcjonujący. Dzieje się tak prawdopodobnie z powodu zbyt skomplikowanego oprogramowania, które wspiera to rozwiązanie. Ostatnio zostało zaproponowane, że wyjściem z obecnego impasu może być zastosowanie agentów programowych (ang. software agents). Niniejsza praca przedstawia propozycję systemu opartego o agentów do zarządzania zasobami w siatce komputerowej.

Agenci Użytkownika działają autonomicznie próbując wykonać powierzone przez użytkownika zadania (ang. jobs) używając dostępnych zasobów siatki, które spełniają określone wymagania. Z drugiej strony, każdy zasób siatki jest reprezentowany przez *Agenta Robotnika*, który oferuje usługi zasobu klientom siatki wedle ustalonej polityki zasobu. System stawia sobie za cel dwa kluczowe wyzwania: (1) siatka przejawia bardzo dynamiczną naturę, gdzie zasoby mogą się pojawiać i znikać bez żadnego ostrzeżenia, (2) natężenie pracy (ang. workload) poszczególnych węzłów jest podatne na nagłe i nieprzewidywalne fluktuacje. Dlatego też, zostały wprowadzone *zespoły agentów*, które są w stanie zapewnić *Jakość Usługi* (ang. Quality of Service) oraz *Umowę Serwisową* (ang. Service Level Agreement). Zespoły te są wirtualnymi organizacjami kolaborujących ze sobą *Agentów Robotników*, na czele których stoi *Agent Przywódca.*

Zespoły Agentowe publikują swoją ofertę w *Centrum Informacji Klienta* (CIK), które jest scentralizowaną żółtą książką (ang. yellow pages) siatki dostępną dla wszystkich klientów siatki poprzez *Agenta CIK.* Oferta zespołu, opisana ontologicznie i przechowywana w relacyjnej bazie danych, stanowi listę oraz opis zasobów oferowanych przez zespół. *CIK* umożliwia *Agentom Użytkownika* odszukanie zespołów, które spełniają wymagania zadania postawione przez użytkownika. Interakcja z *CIK* następuje we wczesnej fazie niemal każdej akcji systemowej, dlatego *CIK* musi być szybki, niezawodny, ponad to powinien udostępniać najbardziej aktualne oferty zespołów. Zostały przygotowane trzy architektury *CIK*: wielowątkowa, wieloagentowa lokalna oraz wieloagentowa rozproszona. Empirycznie otrzymane wyniki wydajnościowe są przedstawione i przeanalizowane. Najlepsze rozwiązanie zostało poddane dalszym modyfikacjom celem zapewnienia jeszcze lepszej wydajności oraz zabezpieczenia niezawodności za pomocą mechanizmu odzyskiwania (ang. recovery mechanism).

Agenci Użytkownika negocjują z potencjalnymi zespołami, otrzymanymi z *CIK*, wykorzystując *Protokół Sieci Kontraktowej FIPA* (ang. FIPA Contract Net). Oferty przedłożone przez zespoły są poddane walidacji względem warunków brzegowych określonych przez użytkownika. Następnie najlepsza oferta jest wybierana za pomocą *Analizy Wielokryterialnej.*

Contents

Chapter 1

Introduction

Grid computing is a promising approach of utilizing geographically distributed, multi-domain, heterogeneous computational resources. Virtualization of computing resources by grid computing is expected to provide its users with highly available and adaptable computing utilities. It is also expected to have a broad impact in science, businesses and industries. It is worth noting spectacular successes of scientific projects like *SETI@Home* and *Folding@Home* utilizing idleness of millions of distributed, internet-connected desktop machines. Although these projects were specific in nature, they showed power of distributed computing – with 900000 computers *SETI@home* has ability to compute 250 TFLOPS, whereas fastest supercomputer (*Blue Gene*) has 280 TFLOPS. The up-take of the grid, while speeding up recently, is still unsatisfactory. One possible reason for this situation is an overly complicated support for resource management provided by current grid software infrastructure.

It has been suggested that software agents with ontologies may provide the necessary infrastructure, by infusing grid with intelligence [1, 2]. Foster *et al.* note that grid community have recently focused on "brain" – sophisticated infrastructure and tools for reliable and secure resource sharing within dynamic and geographically distributed virtual organizations, while software agents community have focused on "brain" – flexible and adaptable problem solvers. It is supposed that grid computing could benefit from autonomous and flexible behaviors of software agents.

As pointed out by Cao *et al.* [3] there are two key challenges that must be addressed:

- **Scalability**. A given component of the grid will have it's own functions, resources, and environment. These are not necessarily geared to work together in the overall grid. They may be physically located in different organizations and may not be aware of each other.

- **Adaptability**. A grid is a dynamic environment where the location, type, and performance of the components are constantly changing. For example, a component may be added to, or removed from, the grid at any time. These resources may not be entirely dedicated to the grid; hence their computational capabilities will vary over time.

The goal of the thesis is to present proposal of the agent-based system for resource management on the grid utilizing ontology demarcated data. Chapter 2 presents overall system design, also published in [4]. Note that the thesis itself is part of the project, and therefore we focus ourselves on the part of the system functionalities, namely in chapter 3 we present *Client Information Center* and in chapter 4 negotiation between *User Agent* and *agent teams*. Work advances related with *Client Information Center* were also reported in [5].

Chapter 2

Proposed system

Grid can be viewed as an environment, in which owners of resources (mostly computational resources) want to offer their resources for usage and be remunerated for it. On the other site, there are users who want to complete their tasks utilizing available grid resources. Approaching this problem in agent-oriented manner, we start building up grid agency, which at the beginning consists of:

- *worker agents* – represent grid resources and act on behalf their owners, autonomously strive to meet its given objectives (e.g. gaining as much profit as possible for resource lease, or contributing to scientific-oriented projects)
- *user agents* – represent and act behalf of users (or whole organizations) that have jobs to be completed utilizing available grid resources, autonomously strive to meet given user expectations (e.g. deadline, budget)

Out most basic assumption is that **single** *worker agent*, for example representing a typical "home-user", can be of little value in real-life open-world grid applications. While we recognize success of applications like *SETI@home* or *Folding@home* that are based on harnessing millions of "home-PCs", these applications are of a 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 fact that the PC processing it "died" can be completed some time in future by another resource. This, however is not the case in realistic (business-type) applications, where calculations have to be completed in a specific order and, what is mostly the case, in a well-defined time-frame. Therefore, to address this problem, we introduce virtual organizations, called *agent teams*. Each team consists of a leader, the *Local Master Agent* (*LMaster*). It is the *LMaster* with whom *user agents* negotiate terms of task execution, and who decides whether accept a new *worker agent* to the team. Throughout the thesis *worker agent* is also referred as *Local Slave Agent* (*LSlave*).

Having *agent teams*, we can impose some level of assurance that task completion will meet user's conditions. We adapt the idea of *Service-Level Agreement* (*SLA*) from [6], which is kind of contract, settled before job execution between *user agent* and team leader – *LMaster*, **imposing** job completion conditions that must be met (e.g. deadline). Furthermore, if for any reason team does not fulfill *SLA* then it may be penalized, paying fine previously included in the agreement. Assuring such an *SLA* in case of a single *worker agent*, representing ordinary home-PC, may be rather difficult as mostly such ordinary machines does not have backup power supplies (like UPS) and thus are likely to fail. In case of *agent teams*, such a situation can be easily overcome by resubmitting execution of the job (or continuation of a job if parts of the results are available) to other *worker agents*.

For an *agent team* to be visible to other agents (potential users or team members), it must post its *team advertisement* in such a way that it will be reachable to other agents. As described in [7], there are many ways in which information that is to be used in matchmaking can be posted in a distributed system and each of them has its advantages and disadvantages. In [8] there has been presented multi-agent system with *yellow page* approach to matchmaking. Based on that work we think that this type of matchmaking can be successfully adapted in our multi-agent system. Thus *LMaster* agents post their team advertisements within the *Client Information Center* (*CIC*). Such an advertisement contains 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* will be used twofold by:

1. *User agents* looking for resources satisfying their task requirements.

2. *Worker agents* searching for a team to join.

For example, *worker agent* representing computational resource with installed MPI software, may want to join a team specializing in solving problems utilizing MPI software. Note that such a 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 on time or almost on time.

In our proposed system, user initiates the execution of the job by providing its *user agent* with specific requirements such as: *resource require-*

ments – required capabilities of resources that are to execute the task, and *execution constraints* – time, budget etc. From this moment on, the *user agent* autonomously acts on behalf of its user. First, it queries the *CIC* for resources matching specified requirements. In response it obtains (from the *CIC*) a list of query-matching *agent teams*. Then it negotiates with *LMasters* representing selected teams, taking into account its *execution constraints* to find the best team for the job.

Similarly, user can specify that its agent joins a team, and specify 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, how can we make this agent-infrastructure as efficient as possible; and this is the focus of this paper. To find answer to this general question, we will introduce a number of possible architectures of the *CIC agent* and its co-workers and empirically establish their performance.

Chapter 3

Client Information Center

Client Information Center (*CIC*) is grid *yellow pages* storing and providing information about available grid resources. Since our system follows general tenets of agent-system design, the *CIC* is implemented as an agent infrastructure consisting of *CIC Agent* and possibly auxiliary agents (discussed in architecture considerations, section 3.3 on page 21). The *CIC Agent* is the agent via which grid clients have access to *CIC* services:

- (de)registering grid resources
- querying of grid resources

Section 3.1 describes *team advertisements* and matchmaking. It is followed by description of interactions with *CIC Agent* in section 3.2 on page 16. Observe that in all situations involving agents interaction with the system, they have to interact with the *CIC* **first**. This being the case, how can we make this agent-infrastructure as efficient as possible. To find answer to this general question, we will introduce number of possible architectures of *CIC Agent* and its co-workers (section 3.3 on page 21) and empirically establish their performance (section 3.4 on page 25).

3.1 Grid yellow pages and matchmaking

For an *agent team* to be visible to other agents (*User Agents*, *Worker Agents*) it must prepare and post its *team advertisement* within *CIC*, which stores the advertisements in *yellow pages*. The incoming resource queries are then executed on the yellow pages. Standard matchmaking takes place here, user resource requirements are matched against *team advertisements* in yellow pages.

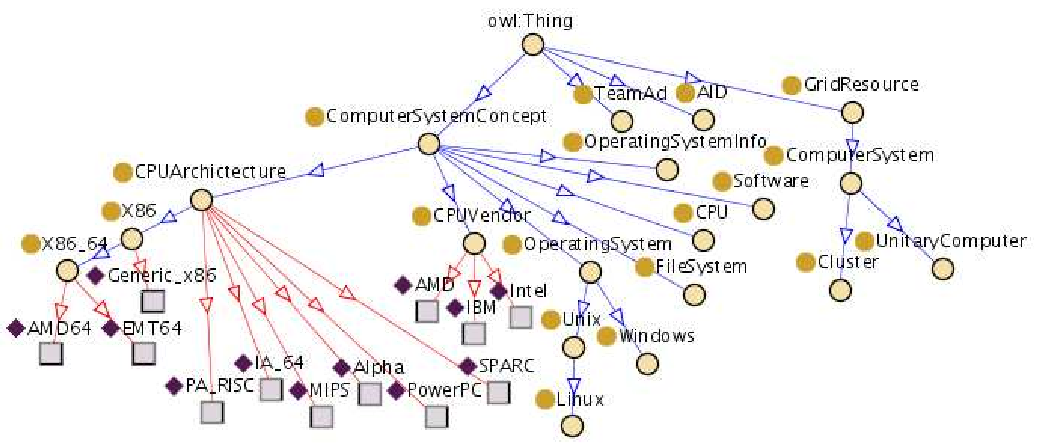


Figure 3.1: Diagram of classes in *Grid Yellow Pages Ontology*. The arrows on diagram show *rdfs:subClassOf* relationships.

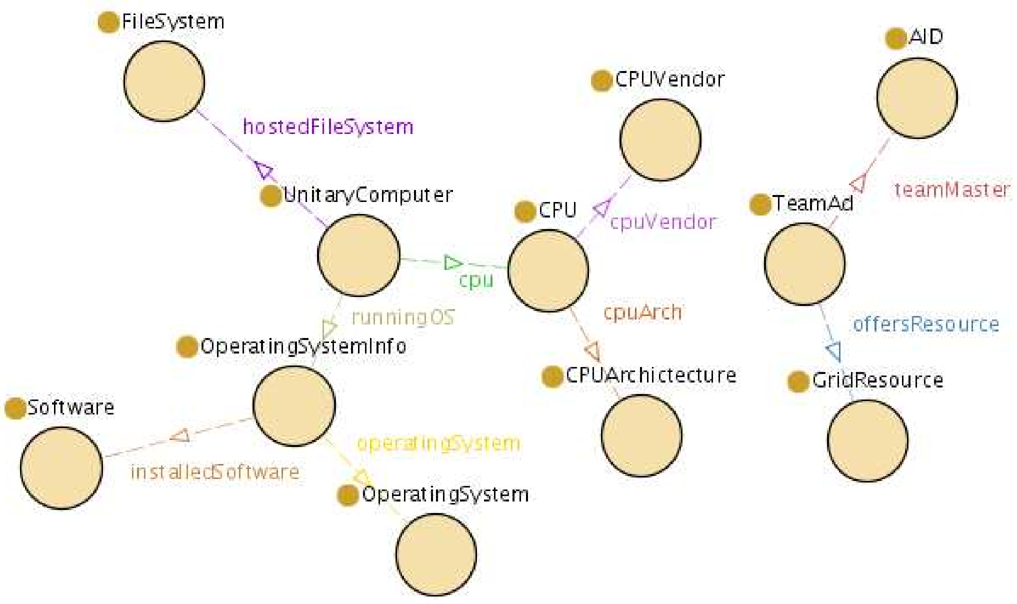


Figure 3.2: Diagram of object properties in *Grid Yellow Pages Ontology*. The arrows on the diagram show *rdfs:domain* and *rdfs:range* relationships.

Team advertisements are expressed and stored as semantically demarcated data according to the *Grid Yellow Pages Ontology* (figure 3.1 to 3.2 on the preceding page). While there exists a number of separate and incompatible attempts at designing an ontology of the grid, at this stage they can be treated only as "work in progress". In other words, "all-agreed" ontology of the grid does not exist. However, we took careful implementation steps in our resource management system, so when there will be standardized grid ontology the system will be ready for it.

We decided to use OWL as an ontology language. There are three OWL sublanguages: OWL Lite, OWL DL, OWL Full (sorted in order of increasing expressiveness). Since there are many operations involving manipulations on ontologies in the system, we found performance of inference and validation engines to be the most important factor while choosing suitable sublanguage. Generally, in case of inference engines, the more expressive sublanguage is the higher worst-case complexity is. Hence, wanted to stay on safe side, we chose OWL Lite. Most of the vocabulary concepts describing grid resources were taken from Common Information Model[1] (CIM). The complete specification of the ontology can be found in appendix A on page 45. Let us analyze example *team advertisement* (expressed in N3 notation[2]):

```
:teamX a YellowPages:TeamAd ;
  yellowPages:teamMaster [
    yellowPages:aidStr "monster@e-plant.com:1099/JADE"
  ] .
  yellowPages:offersResource :PC2929 .
  yellowPages:offersResource :PC2930 .

:PC2929 a yellowPages:UnitaryComputer ;
  yellowPages:totalCPUCount "1" .
  yellowPages:cpu [
    yellowPages:cpuVendor yellowPages:Intel ;
    yellowPages:cpuArch yellowPages:Generic_x86 ;
    yellowPages:cpuClockSpeed "1500" ;
    yellowPages:cpuCount "1"
  ] .
  yellowPages:hostedFileSystem [
    yellowPages:availableSpace "8000"
  ] .
```

[1] *Grid Yellow Pages Ontology* is based on Computer System, Processors, Local File Systems and Operating System models defined in Common Information Model (CIM) [9].

[2] Notation 3 (N3) is a language which is compact and readable alternative to RDF's XML, see specification: http://www.w3.org/DesignIssues/Notation3.html

```
  yellowPages:runningOS [
    yellowPages:freePhysicalMemory "1000" ;
    yellowPages:freeVirtualMemory "2000" ;
    yellowPages:operatingSystem [
      a yellowPages:Linux ;
      yellowPages:osType "Debian Linux" ;
      yellowPages:osVersion "3.1"
    ] ;
    yellowPages:installedSoftware [
      yellowPages:softwareName "JRE" ;
      yellowPages:softwareVersion "1.4.2"
    ] ;
  ] .

:PC2930 a yellowPages:UnitaryComputer ;
  yellowPages:totalCPUCount "1" .
  yellowPages:cpu [
    yellowPages:cpuVendor yellowPages:Intel ;
    yellowPages:cpuArch yellowPages:Generic_x86 ;
    yellowPages:cpuClockSpeed "2000" ;
    yellowPages:cpuCount "1"
  ] .
  yellowPages:hostedFileSystem [
    yellowPages:availableSpace "60000"
  ] .
  yellowPages:runningOS [
    yellowPages:freePhysicalMemory "1000" ;
    yellowPages:freeVirtualMemory "2000" ;
    yellowPages:operatingSystem [
      yellowPages:osType "SuSE Linux" ;
      yellowPages:osVersion "10.0"
    ] ;
    yellowPages:installedSoftware [
      yellowPages:softwareName "JRE" ;
      yellowPages:softwareVersion "1.4.2"
    ] ;
  ] .
```

Above *team advertisement* describes “teamX” whose *LMaster* is an agent identified by “monster@e-plant.com:1099/JADE” which is JADE Agent IDentifier (AID) needed for an agent to be able to contact team master within agent platform. Team offers two computational resources (unitary computers) with the following capabilities:

- PC2929
  - processor: Intel, 1500MHz clock speed, x86 architecture
  - 8GB file system space
  - 1GB of free physical memory and 2GB of free virtual memory
  - operating system: Debian Linux 3.1
  - installed Java Runtime Environment (JRE) 1.4.2
- PC2930
  - processor: Intel, 2000MHz clock speed, x86 architecture
  - 60GB file system space
  - 1GB of free physical memory and 2GB of free virtual memory
  - operating system: SuSE Linux 10.0
  - installed Java Runtime Environment (JRE) 1.4.2

Matchmaking To query semantically demarcated yellow pages we utilize SPARQL (Query Language for RDF) [10]. Let us now assume that *User Agent* is looking for two machines with processor of at least 1GHz, at least 256MB of RAM and installed JRE. Additionally one machine should have at least 10GB of disk space for storing some data. Then the SPARQL query will have the form:

```
PREFIX yellowPages:
  <http://gridagents.sourceforge.net/YellowPages#>
SELECT DISTINCT ?teamContact
WHERE {
  ?team yellowPages:teamMaster [
    yellowPages:aidStr ?teamContact
  ] .
  ?team yellowPages:offersResource ?res1 .
  ?team yellowPages:offersResource ?res2 .
  FILTER (?res1 != ?res2)

  # Matching first machine
  ?res1 yellowPages:cpu [
    yellowPages:cpuClockSpeed ?cpuClockSpeed1 ;
  ] .
  FILTER (?cpuClockSpeed1 > 1000)
  ?res1 yellowPages:runningOS [
    yellowPages:freePhysicalMemory ?freePhysicalMem1 ;
```

```
      yellowPages:installedSoftware [
        yellowPages:softwareName ?softwareName1 ;
      ]
    ] .
    FILTER (?freePhysicalMem1 > 256 && ?softwareName1 = "JRE")

` # Matching second machine
    ?res2 yellowPages:cpu [
      yellowPages:cpuClockSpeed ?cpuClockSpeed2 ;
    ] .
    FILTER (?cpuClockSpeed2 > 1000)
    ?res2 yellowPages:runningOS [
      yellowPages:freePhysicalMemory ?freePhysicalMem2 ;
      yellowPages:installedSoftware [
        yellowPages:softwareName ?softwareName2 ;
      ]
    ] .
    FILTER (?freePhysicalMem2 > 256 && ?softwareName2 = "JRE")
    ?res2 yellowPages:hostedFileSystem [
      yellowPages:availableSpace ?availSpace2
    ]
    FILTER (?availSpace > 10000)
}
```

The above query matches team advertisement (described at the beginning of the section) and returns "monster@e-plant.com:1099/JADE", which is AID of master agent of "teamX". Obviously, the complete response would consist of a list of all teams that have among them at least two machines that satisfy the criteria described above.

3.2 Interaction with CIC Agent

Grid clients can request *CIC Agent* to perform an action (e.g. query) simply by sending an appropriate request message to it. The intercommunication between agents is standardized by *CIC Request Protocol*, which is fully compliant with *FIPA Request Protocol*[3]. Figure 3.3 on the next page[4] depicts interaction between *requestor* and *CIC Agent*. *Requestor* can be either *User Agent*, querying for available grid resources, or *LMaster*, registering/updating *team advertisement*. First, *requestor* sends appropriate request message,

[3]See `http://fipa.org/specs/fipa00026/SC00026H.pdf` for *FIPA Request Interaction Protocol* specification

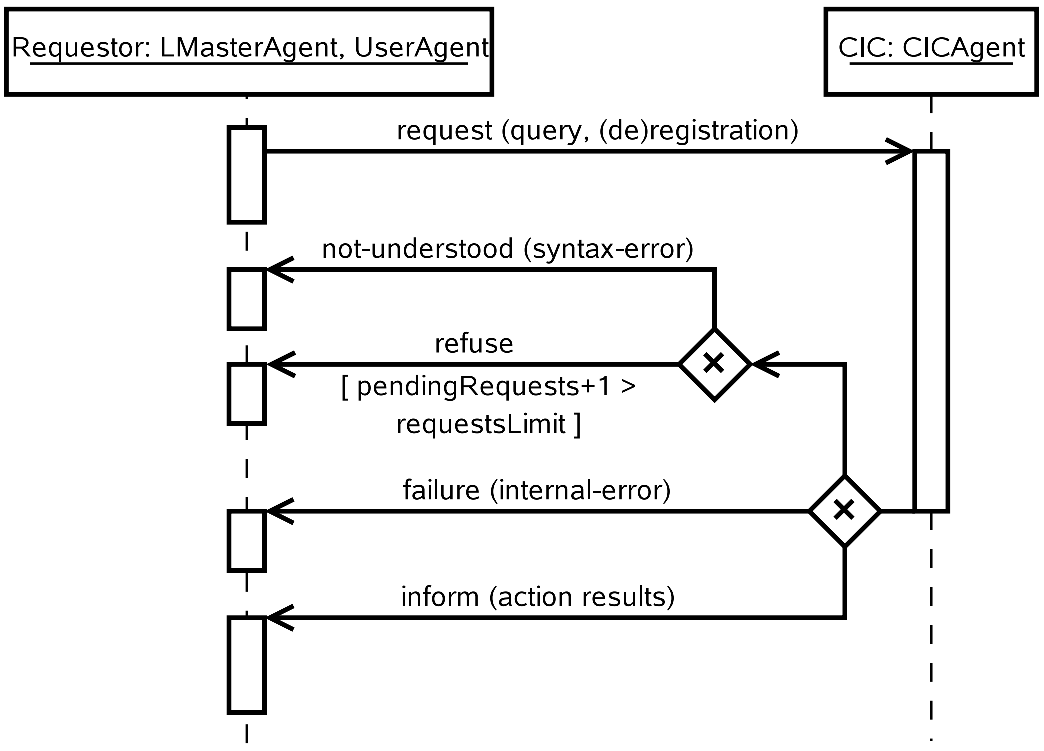

Requestor: LMasterAgent, UserAgent
CIC: CICAgent
request (query, (de)registration)
not-understood (syntax-error)
refuse
[pendingRequests+1 > requestsLimit]
failure (internal-error)
inform (action results)

Figure 3.3: Interaction protocol diagram for *CIC Request Protocol.*

then *CIC Agent* replies with one of the following *communicative acts*:

- *not-understood*, if the contents of the message could not be understood, more specifically, it could not be successfully extracted according to *Messaging* ontology. The problem lies at *requestor's* site.
- *refuse*, *CIC* has the limit of pending requests and thus may refuse to perform another request if the request queue is full. *Requestor* should try later on.
- *failure*, if for any reasons *CIC* fails to service a request (for example, database is down). The problem lies at *CIC* site.
- *inform*, if request has been completed successfully. Depending on the kind of request, it may be either *inform-done*, simply denoting that request has been completed successfully, or *inform-result*, which contains action results in message contents.

In agent world the interaction between agents takes place by exchanging messages. The syntactic conformance is forced by usage of FIPA Semantic Language (SL)[5]. For the communication to be meaningful agents must have the same semantic understanding of the message contents, therefore we introduce *Messaging Ontology* that describes agent actions, their results and concepts that can be utilized during interaction. Figure 3.4 on the facing page depicts *Messaging Ontology* (see appending B on page 50 for complete specification of *Messaging Ontology*).

Example interaction There is *User Agent* which queries *CIC Agent* to describe resources that have CPU clock speed greater then 1000MHz. The following two messages are exchanged in this example scenario:

```
(REQUEST
  :sender (agent-identifier
    :name tester@kameleon:1099/JADE
    :addresses (sequence http://kameleon:7778/acc )
    :X-JADE-agent-classname gridrm.test.cic.CICTesterAgent
  )
 :receiver (
    set (agent-identifier
      :name cic-agent@kameleon:1099/JADE
      :addresses (sequence http://kameleon:7778/acc)
    )
  )
```

[4]*Interaction protocol diagram* was introduced in Agent Unified Modeling Language (AUML), see [11, 12] for details.

[5]See http://fipa.org/specs/fipa00008/SC00008I.html for FIPA Semantic Language specification.

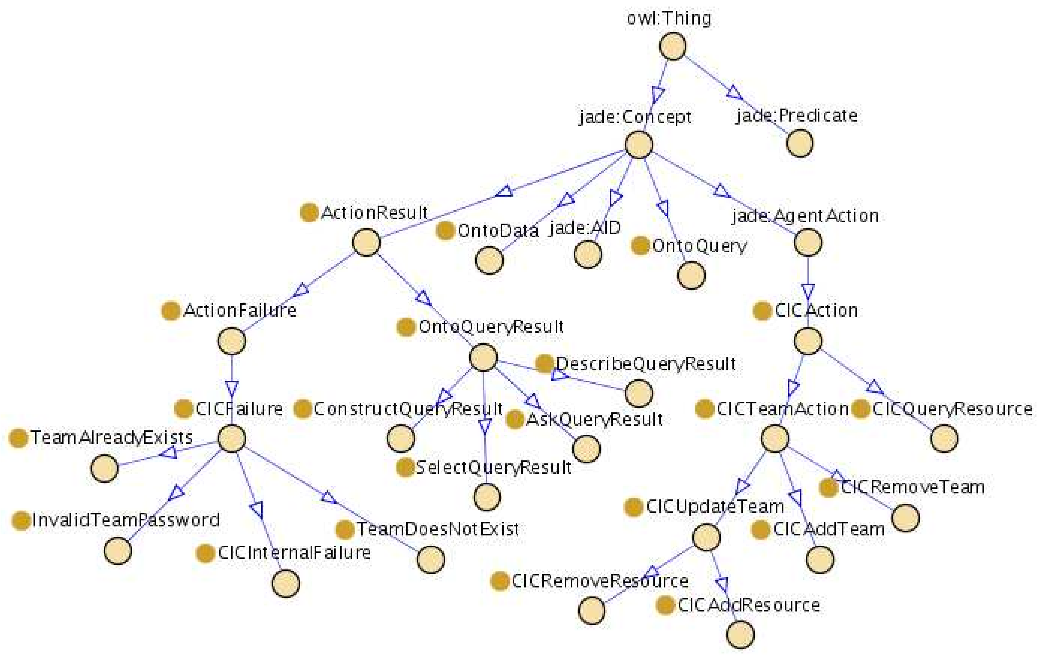


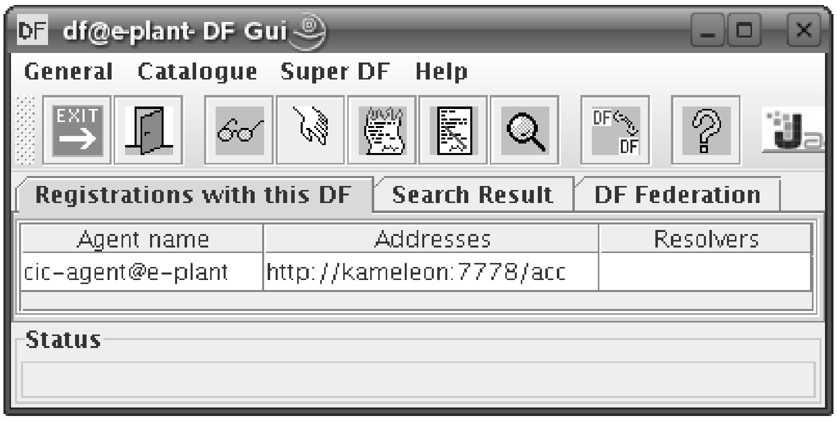
Figure 3.4: Class diagram of *Messaging Ontology*. The arrows show *rdfs:subClassOf* relationship between classes.

```
 :content "(
    (action
      (agent-identifier
        :name cic-agent@kameleon:1099/JADE
        :addresses (sequence http://kameleon:7778/acc)
      )
      (CICQueryResource
        :resourceQuery (OntoQuery
          :ontoQuerySyntax http://jena.hpl.hp.com/2003/07/query/SPARQL
          :ontoQueryStr
            \"PREFIX grid: <http://gridagents.sourceforge.net/YellowPages#>
              DESCRIBE ?res
              WHERE { ?res grid:cpu [ grid:cpuClockSpeed ?cpuSpeed ] .
              FILTER (?cpuSpeed >= 1000)}\"
        )
      )
    )
  )"
 :language  fipa-sl0
 :ontology  Messaging
 :reply-by  20060501T145408274Z
 :protocol  fipa-request
 :conversation-id  C26811873_114649524528
)

# reply from CIC Agent:

(INFORM
  :sender (agent-identifier
      :name cic-agent@kameleon:1099/JADE
      :addresses (sequence http://kameleon:7778/acc )
      :X-JADE-agent-classname gridrm.agents.cic.CICAgent
    )
    :receiver (set
      (agent-identifier
        :name tester@kameleon:1099/JADE
        :addresses (sequence http://kameleon:7778/acc )
        :X-JADE-agent-classname gridrm.test.cic.CICTesterAgent
      )
    )
  :content "(
      (result
        (action
          (agent-identifier
            :name cic-agent@kameleon:1099/JADE
            :addresses (sequence http://kameleon:7778/acc)
          )
          (CICQueryResource
            :resourceQuery
              (OntoQuery
                :ontoQuerySyntax http://jena.hpl.hp.com/2003/07/query/SPARQL
                :ontoQueryStr
                  \"PREFIX grid:
                      <http://gridagents.sourceforge.net/YellowPages#>
                    DESCRIBE ?res
                    WHERE { ?res grid:cpu [ grid:cpuClockSpeed ?cpuSpeed ] .
                    FILTER (?cpuSpeed >= 1000)}\"
              )
          )
        )
        (RdfGraphResult
          :rdfGraph (OntoData
            :ontoDataLang RDF/XML-ABBREV
            :ontoDataStr
\"<rdf:RDF
        xmlns:grid="http://gridagents.sourceforge.net/YellowPages#"
        xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
    <grid:UnitaryComputer rdf:about="jade://myLaptop@kameleon:1099/JADE">
      <rdf:type rdf:resource=
        "http://gridagents.sourceforge.net/Grid#ComputerSystem"/>
      <grid:cpu>
        <grid:cpuClockSpeedMhz
            rdf:datatype="http://www.w3.org/2001/XMLSchema#int">
          1500
        </grid:cpuClockSpeedMhz>
      </grid:cpu>
    </grid:UnitaryComputer>
  </rdf:RDF>\"
          )
        )
      )
    )"
  :reply-with  tester@kameleon:1099/JADE1146495245939
  :language  fipa-sl0
  :ontology  Messaging
  :protocol  fipa-request
  :conversation-id  C26811873_1146495245282
)
```

Localization of CIC Agent Each grid agent must be able to locate *CIC Agent*. Therefore we use agent platform's *yellow pages* service provided by *Directory Facilitator* (*DF*) agent. Once *CIC Agent* is ready to service clients it registers within *DF* (figure 3.5), so other agents can locate it. There are two service types that are registered: *cic-resource-querying* and *cic-resource-registration* (listings 3.1 to 3.2 on pages 21–22 respectively). Whenever grid agents want to contact *CIC*, they query *DF* for agent that is offering one of the service types depending on what they want to do. Description of the services explicitly states that interaction should be initiated according to *FIPA Request Protocol*, message contents should be encoded using minimal subset of *FIPA SL* codec and *Messaging* ontology.



Agent name	Addresses	Resolvers
cic-agent@e-plant	http://kameleon:7778/acc	

Figure 3.5: *CIC Agent* registered in *DF*.

```
(service-description
   :name       "cic"
   :type       "cic-resource-querying"
   :protocol   (set "FIPA-Request")
   :ontology   (set "Messaging")
   :language   (set "FIPA-SL0")
)
```

Listing 3.1: Description of *cic-resource-querying* service

3.3 Architectures

The *CIC* infrastructure is one of key components in our system. Therefore, it must be reliable, fast, and capable of efficiently handling large number of

```
(service-description
   :name       "cic"
   :type       "cic-resource-registration"
   :protocol  (set "FIPA-Request")
   :ontology  (set "Messaging")
   :language  (set "FIPA-SL0")
)
```

Listing 3.2: Description of *cic-resource-registration* service

requests. Specifically, since interactions between *User Agents* and the *CIC* are the key part of early stages of job execution, or *Worker Agents* joining an agent team, long delays in *CIC* responses would become a major bottleneck of the whole system.

In this context, let us note first, that in our system the *yellow page* information is stored in an ontologically demarcated form. To facilitate it we use Jena 2.3 [13] and its database persistency mechanism (see [14] for a report on using Jena with massive store of ontological triples). In [15] it has been 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 should 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 3.6.

2. multi-agent *CIC* with local database agents (*CICDbAgents*), see figure 3.7.

3. multi-agent *CIC* with distributed database agents – located on separate computers (based on the idea from [15]), see figure 3.8

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

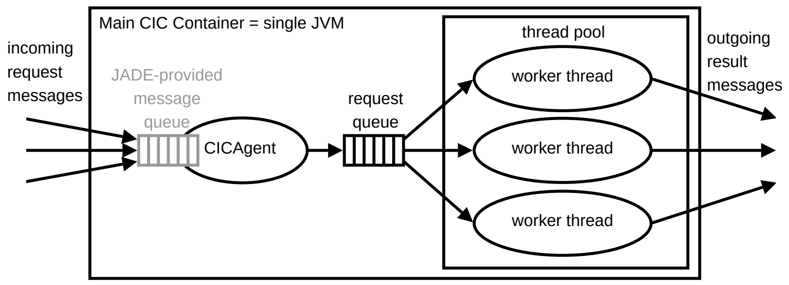


Figure 3.6: Request/result flow in a multi-threaded *CIC* architecture.

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

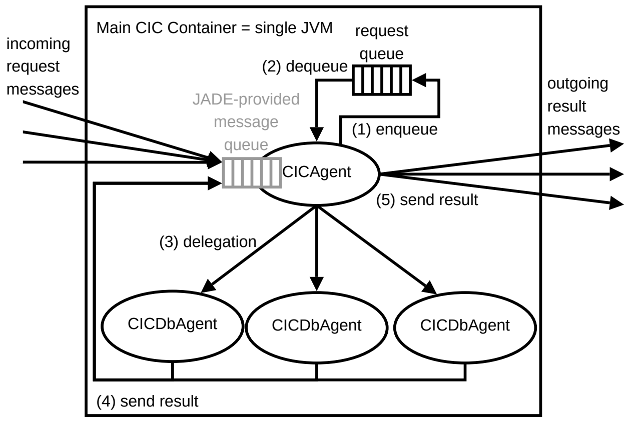


Figure 3.7: Request/result flow in multi-agent *CIC* architecture with local *CICDbAgents*.

In the second approach we use local (residing in the same agent container) *CICDbAgents*—instead of worker threads. The *CIC* agent picks requests form the JADE *message queue* and enqueues them into the 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 *CIC Agent*. As a result they are placed in the same JADE *message queue* as the incoming query-requests. There are two behaviors within the *CIC Agent* that are servicing the JADE *message queue*. One of these behaviors 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 this has very important consequences for the performance of this approach.

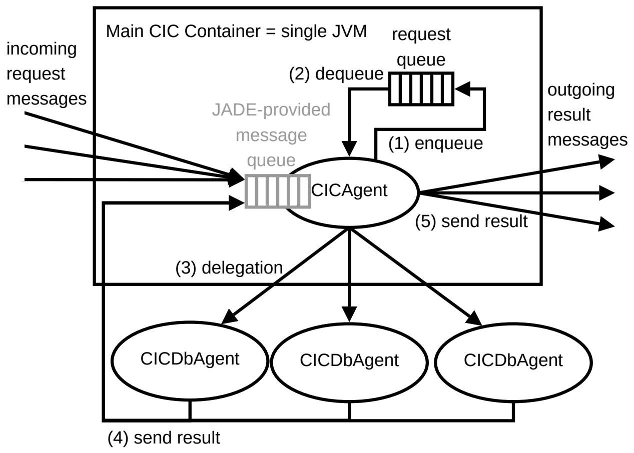


Figure 3.8: Messages flow in multi-agent *CIC* architecture with distributed *CICDbAgents*.

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*.

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.”

3.4 Performance experiments

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 in Jena, responsible for executing the query on RDF resources persisted in database, translates only parts of the SPARQL query into SQL. The rest (e.g. FILTER operations) of the query are performed not by the database itself, 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)
  ?master grid:offersResource ?comp
}
```

Each *QA* was running concurrently on 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 end of the experiment

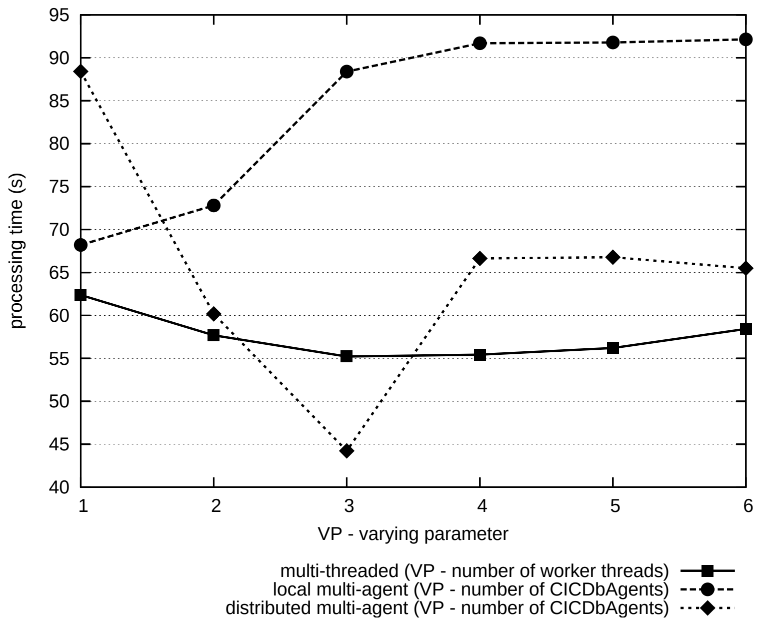


Figure 3.9: Experimental results of different *CIC* architectures: multi-threaded, multi-agent with local *CICDbAgents* and multi-agent with distributed *CICDbAgents*; processing time of 10, 000 queries depending on number of worker threads in multi-threaded architecture/number of *CICDbAgents* in multi-agent architectures.

Results At figure 3.9 we represent total processing time of 10, 000 requests by each *CIC* architecture, when the number of agents/worker threads increases from 1 to 6. The figures 3.10 to 3.12 on pages 27–28 show the best throughputs achieved in local multi-agent, distributed multi-agent and multi-threaded architectures respectively. 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 are used.

What is somewhat surprising is the fact that the architecture with local *CICDbAgents* does not scale well. 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

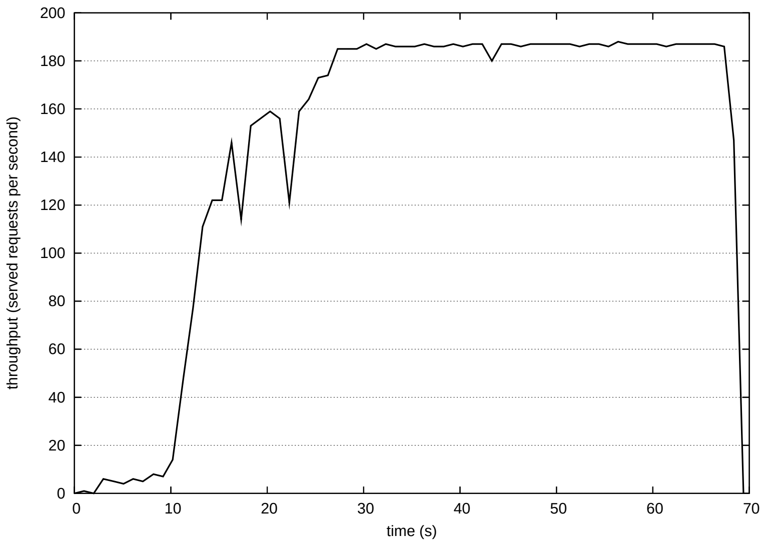


Figure 3.10: Throughput of multi-agent architecture with 1 local *CICD-bAgent*.

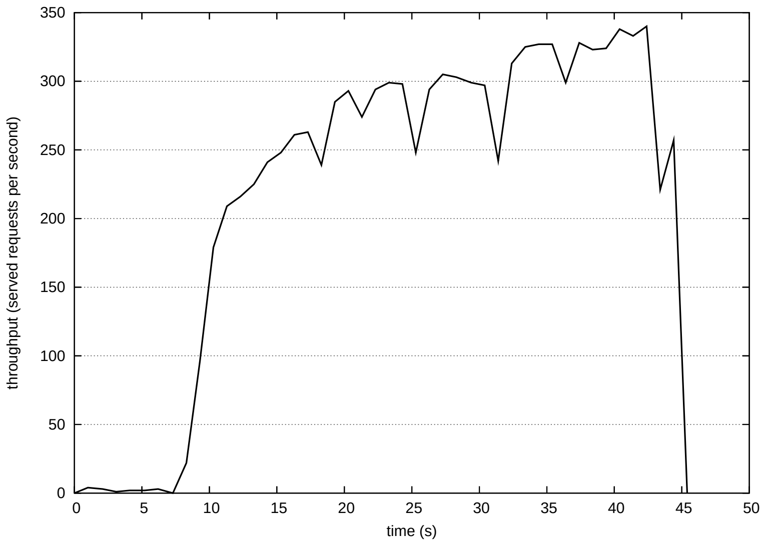


Figure 3.11: Throughput of multi-agent *CIC* architecture with 3 *CICD-bAgents*.

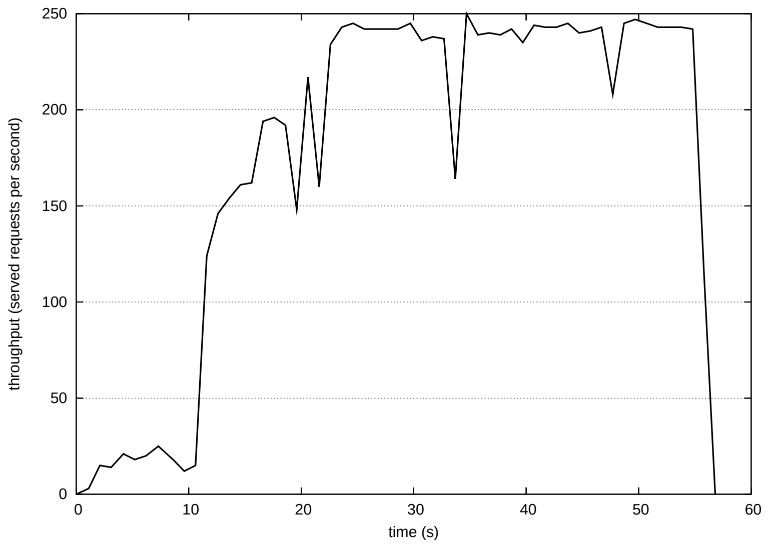


Figure 3.12: Throughput of multi-threaded *CIC* architecture with 3 worker threads.

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 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. Note also that this situation cannot be changed programmatically, since this is how JADE works internally. 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 3.16 (even though this figure represents performance of the last approach, behavior noted in the case of local agents was of a very similar nature). There we can observe that, as the number of request messages decreases, throughput increases. Obviously, this effect is somewhat biased by the way that our experiment was set-up; all query-request messages were sent at once at the beginning and then *Querying Agents* were just waiting for responses.

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 increases past 3, the same effect as in the case of local agents – performance decrease caused by the way that the *CICAgent* removes data from the queue – can be observed.

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

Performance tuning Based on observations collected thus far we have decided to attempt at improving the performance of the third *CIC* architecture – the multi-agent 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 3.13). This agent

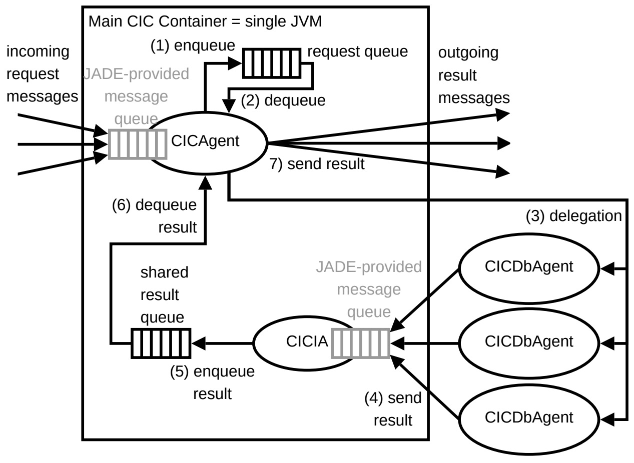


Figure 3.13: Request/result flow in distributed multi-agent *CIC* with *CICIA*.

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 *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 JADE *message queue* of the *CICIA*). Upon reception of such messages, *CICIA* enqueues them into a synchronized *result-queue* (that we have implemented), 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 message from the JADE message request 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 3.14. 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 message queue.

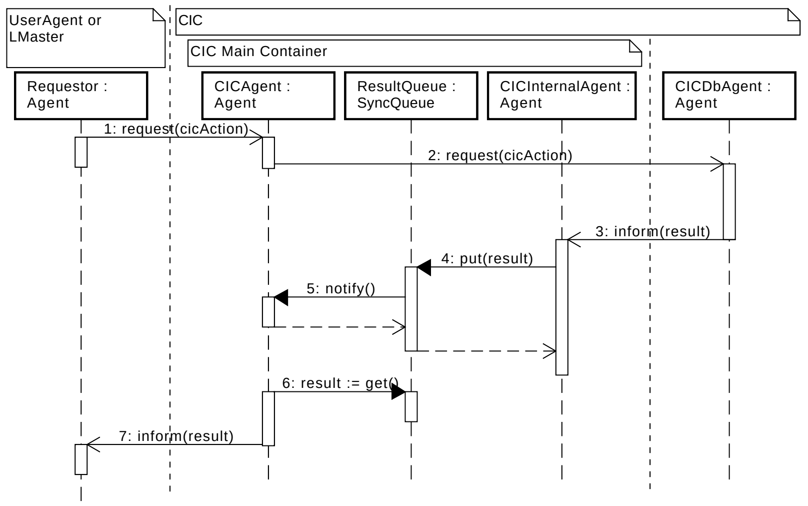


Figure 3.14: Sequence diagram: Handling requests by *CIC* with additional *CIC Internal Agent.*

In figure 3.15 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 only

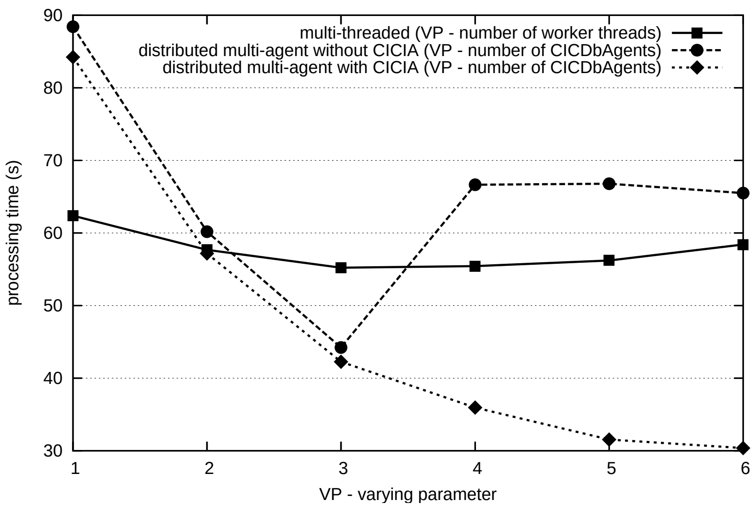


Figure 3.15: Comparison of multi-threaded *CIC* with distributed multi-agent architecture with and without *CIC Internal Agent*; processing time of 10,000 queries varying number of worker threads/number of *CICDbAgents*.

slightly better). However, as the number of *CICDbAgents* increases up to 6, the performance continues to improve steadily. Since we can observe also a leveling-off effect, the fact that we were not able to run experiments with more than 11 computers (and thus the largest number of *CICDbAgents* was 6), is 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 the worker threads based implementation of the *CIC* is of order 45% (here we compare the best threaded solution – with 3 threads – with that of the distributed agent solution with *CICIA*, for 6 *CICDbAgents*).

Finally, in figure 3.16 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* 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 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.

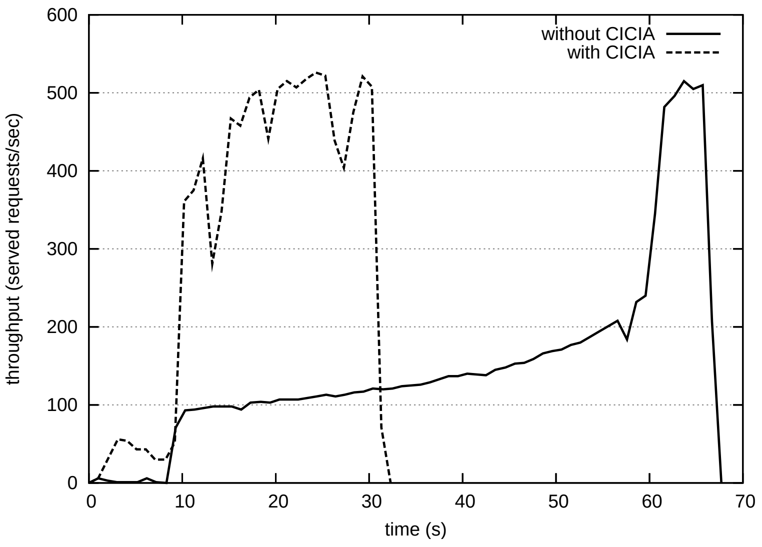


Figure 3.16: Comparison of distributed multi-agent architecture (6 *CICDbAgents*) with and without *CIC Internal Agent.* Throughput (served requests per second) vs. time.

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.

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 we do not utilize remote *CICDbAgents* fully, they are 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%.

3.5 Auxiliary topics

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, (2) the other 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 the agent) without matching results (in the second snapshot). These requests are put back to the request queue for subsequent execution.

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 change implementation of *request queue* into *priority request queue*, which queues requests according to their priority, and thus *CICAgent* strives to delegate 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.

Chapter 4

Job execution from user perspective

The first step to execute a job on the grid is to provide *User Agent* with the necessary information such as job description, negotiation parameters and constraints. Having user input, *User Agent* acts autonomously trying to execute job utilizing available resources meeting user expectations. First it queries *Client Information Center* (*CIC*) for *agent teams* that have required resources for the job. Then it starts negotiation process with team leaders (*LMasters*) taking into account negotiation parameters and constraints specified by the user. The best team is chosen using Multi Criteria Analysis (MCA) [16].

4.1 User Input

The user provides *User Agent* with the job description, negotiation parameters and execution constraints. Job description contains resource requirements expressed in the *Grid Yellow Pages Ontology* (see section 3.1 on page 11). There are three negotiation parameters: cost, job start time and job end time. For each of theses parameters the user specifies its importance by giving weight that is used in MCA (section 4.3 on page 36). For example user may state that cost of the execution is twice important than job end time by giving weight 2 to cost and weight 1 to job end time. If any of theses parameters should not be taken into account then 0 weight is given. The user may also specify constraints such as maximum cost, maximum job start time and maximum job end time. The offers that do not meet these constraints are not taken into account during negotiation phase. We have implemented User Agent GUI as the way of providing *User Agent* with user input (see example in section 4.4 on page 37).

4.2 Negotiation

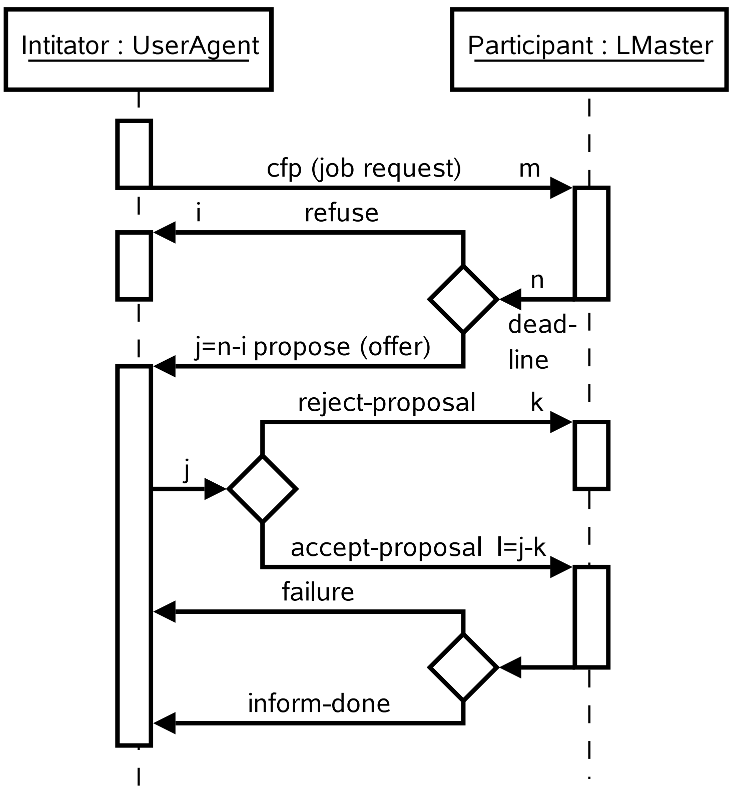


Figure 4.1: Interaction Diagram of FIPA Contract Net Protocol.

After receiving a list of potential teams that are capable of doing the job, *User Agent* starts negotiation with them utilizing FIPA Contract Net Protocol (figure 4.1)[1]. First is sends *CALL-FOR-PROPOSAL* (*CFP*) message to all matched *LMasters*. *CFP* contains job description according to which *LMasters* are able to construct their offers. Once the offer is ready it is sent back to *User Agent* via *PROPOSE* message. It is possible that some of *LMasters* refuse to propose, for example in the meantime some of the resources "disappeared" and now they do not have required resources. The *User Agent* awaits for *PROPOSALs* and *REFUSALs* until all of them are received or deadline occurs. It is important to introduce deadline to avoid

[1]See http://www.fipa.org/specs/fipa00029/SC00029H.html for FIPA Contract Net Interaction Protocol Specification.

too long latencies in the process. If there is no proposal then *User Agent* fails with the task and reports back to the user. Otherwise, if there is at least one *PROPOSAL* then *User Agent* starts evaluation of offers, which is two-stage process:

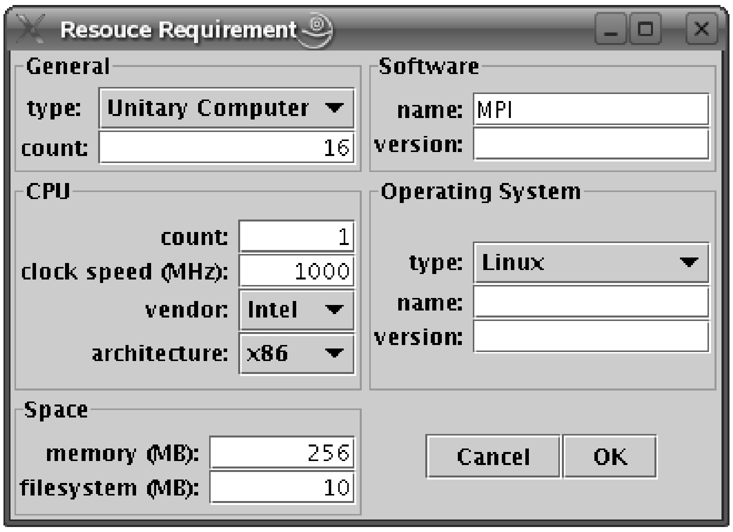
- Offers which do not meet execution constraints (e.g. cost, job start time, job end time) are filtered out and are not taken into account in the next stage. If all offers are filtered out in this stage due to constraints then *User Agent* fails with the task and reports back to the user.

- The remaining offers are evaluated using Multi-Criteria-Analysis module (MCA) – see section 4.3.

After MCA phase there is chosen team for job execution and *ACCEPT-PROPOSAL* is sent to that team. The other teams are rejected by sending *REJECT-PROPOSAL* to them. The chosen team confirms acceptance by *INFORM-DONE* message.

4.3 Multi Criteria Analysis

We use *linear additive model* [16] as Multi Criteria Analysis. It is done by multiplying value scores on each criterion by the weight of that criterion, and then adding all those weighted scores together. We have three criterions that take part in MCA process: cost, job start time and job end time. If there are n teams then criterion scores of i-th team are calculated in the following way:
Start Time Score: $STS_i = \frac{(\frac{1}{startTime_i - currentTime})}{\sum_{j=1}^{n}(\frac{1}{startTime_j - currentTime})}$
End Time Score: $ETS_i = \frac{(\frac{1}{endTime_i - currentTime})}{\sum_{j=1}^{n}(\frac{1}{endTime_j - currentTime})}$
Cost Score: $CS_i = \frac{(\frac{1}{cost_i})}{\sum_{j=1}^{n}(\frac{1}{cost_j})}$
All scores are normalized and generally the better criterion value the higher score it is given. Overall i-th team score calculations:
Team Score: $TS_i = STS_i * startTimeWeight + ETS_i * endTimeWeight + CS_i * costWeight$
Team with the highest overall score, as a weighted sum of criterion scores of the team, is chosen as the winner team. For the example of MCA in use please refer to the section 4.4 on the facing page.


Resouce Requirement

General
type: Unitary Computer
count: 16

Software
name: MPI
version:

CPU
count: 1
clock speed (MHz): 1000
vendor: Intel
architecture: x86

Operating System
type: Linux
name:
version:

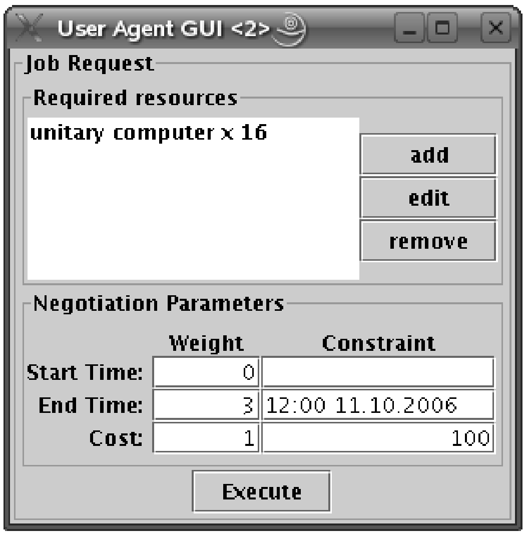
Space
memory (MB): 256
filesystem (MB): 10

Figure 4.2: User Agent GUI: Resource requirements.

4.4 Example

We would like to compute some job utilizing MPI library on 16 machines. First, we specify resource requirements as shown on the figure 4.2. Then we provide negotiation parameters expressing our execution preferences. For example, we must meet deadline of 12:00 11th October 2006, and the cost does not matter as much as time (figure 4.3 on the following page). Therefore we specify deadline as end time constraint and we prefer three times more sooner end time then cheaper cost. We do not care when the job starts (weight 0), we only want to meet specific deadline.

Figure 4.4 on the next page shows matched teams and their scores evaluated by MCA. Let us note that teamA has been rejected because it does not meet deadline constraint. The other teams are calculated score. Despite of cheaper cost 40% of teamC, it was teamB that was accepted to do a job because of sooner end time.



User Agent GUI <2>

Job Request

Required resources

unitary computer x 16

add

edit

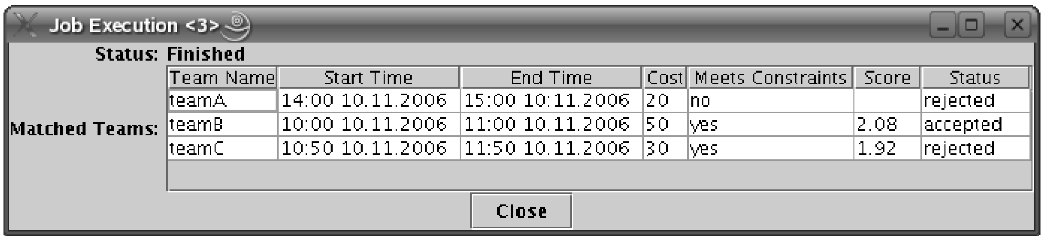
remove

Negotiation Parameters

	Weight	Constraint
Start Time:	0	
End Time:	3	12:00 11.10.2006
Cost:	1	100

Execute

Figure 4.3: User Agent GUI: Weights and constraints of criterions.



Job Execution <3>

Status: Finished

Matched Teams:

Team Name	Start Time	End Time	Cost	Meets Constraints	Score	Status
teamA	14:00 10.11.2006	15:00 10:11.2006	20	no		rejected
teamB	10:00 10.11.2006	11:00 10.11.2006	50	yes	2.08	accepted
teamC	10:50 10.11.2006	11:50 10.11.2006	30	yes	1.92	rejected

Close

Figure 4.4: User Agent GUI: Matched teams and their scores.

Chapter 5

Conclusions

We presented overall vision of agent-based approach to resource management of the grid. It seems that nature of software agents as autonomous software entities striving to meet their objectives proves to be useful in highly dynamic environments such as grids. We have prototyped *Client Information Center* as the grid yellow pages within which agent teams post their team advertisements and *User Agents* look up resources. Expressing *team advertisements* as ontology demarcated data is especially promising in vast environments were common knowledge understanding must be imposed, however no standardized ontology have been worked out by grid community so far. Reliability and performance were addressed as the key challenges in the implementation of *CIC* component. Therefore we prepared three *CIC* architectures and empirically established their performance. It turned out that multi-threaded *CIC* is the best choice for single machine infrastructure, however distributed multi-agent approach seems to be much more efficient. We performed further performance tuning overcoming the observed shortcoming of the agent-platform's message queue by introduction of additional agent – *CICIA*. That architecture outperformed other architectures with the steady throughput of 400–500 requests per second. The reliability of the distributed *CIC* architecture was also taken into consideration by introducing simple recovery mechanism in case of *CICDbAgent* failure. Despite of this, *CIC* remains single point of the failure in the system, if the main *CIC* agent crashes then the whole infrastructure breaks down as well. This critical issue should be considered in the future works. Then we prototyped negotiation between *User Agent* and *agent teams* utilizing FIPA Contract Net Protocol and Multi Criteria Analysis implemented using linear additive model. That part of the system is highly abstract and is only the basis for the future work, we have covered neither the issues of job submission mechanism nor the estimation of the job execution.

Bibliography

[1] Ian Foster, Nicholas R. Jennings, and Carl Kesselman. Brain meets brawn: Why grid and agents need each other. In *AAMAS '04: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 8–15, Washington, DC, USA, 2004. IEEE Computer Society.

[2] Huaglory Tianfield and Rainer Unland. Towards self-organization in multi-agent systems and grid computing. *Multiagent and Grid Systems*, 1(2):89–95, 2005.

[3] Junwei Cao, Darren J. Kerbyson, and Graham R. Nudd. Use of agent-based service discovery for resource management in metacomputing environment. In *Euro-Par '01: Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing*, pages 882–886, London, UK, 2001. Springer-Verlag.

[4] Mateusz Dominiak, Wojciech Kuranowski, Maciej Gawinecki, Maria Ganzha, and Marcin Paprzycki. Utilizing agent teams in grid resource management - preliminary considerations. In *Proceedings of the J. V. Atanasov COnference*, October 2006.

[5] Mateusz Dominiak, Wojciech Kuranowski, Maciej Gawinecki, Maria Ganzha, and Marcin Paprzycki. Efficient matchmaking in an agent-based grid resource brokering system. In *XXII Autumn Meetings of Polish Information Processing Society, to appear*, November 2006.

[6] D. Ouelhadj, J. Garibaldi, J. MacLaren, R. Sakellariou, K. Krishnakumar, and Amnon Meisels. A multi-agent infrastructure and a service level agreement negotiation protocol for robust scheduling in grid computing. In *Advances in Grid Computing - EGC 2005*, volume 3470/2005 of *Lecture Notes in Computer Science*, pages 651–660, Germany, 2005. Springer Verlag.

[7] David Trastour, Claudio Bartolini, and Chris Preist. Semantic web support for the business-to-business e-commerce lifecycle. In *WWW '02: Proceedings of the 11th international conference on World Wide Web*, pages 89–98, New York, NY, USA, 2002. ACM Press.

[8] Costin Badica, Adriana Badita, Maria Ganzha, and Marcin Paprzycki. Developing a model agent-based e-commerce system. In *Jie Lu et. al. (eds.) E-Service Intelligence - Methodologies, Technologies and Applications (to appear)*, 2006.

[9] Common Information Model (CIM) Standards. http://www.dmtf.org/standards/cim.

[10] SPARQL Query Language for RDF. http://www.w3.org/tr/rdf-sparql-query.

[11] James Odell, H. Van Dyke Parunak, and Bernhard Bauer. Representing agent interaction protocols in uml. In *First international workshop, AOSE 2000 on Agent-oriented software engineering*, pages 121–140, Secaucus, NJ, USA, 2001. Springer-Verlag New York, Inc.

[12] Bernhard Bauer, Joerg P. Mueller, and James Odell. Agent uml: a formalism for specifying multiagent software systems. In *First international workshop, AOSE 2000 on Agent-oriented software engineering*, pages 91–103, Secaucus, NJ, USA, 2001. Springer-Verlag New York, Inc.

[13] Jena A Semantic Framework for Java. http://jena.sourceforge.net.

[14] Katie Portwin and Priya Parvatikar. Building and managing a massive triple store: An experience report. http://xtech06.usefulinc.com/schedule/paper/18.

[15] Krzysztof Chmiel, Dominik Tomiak, Maciej Gawinecki, Pawel Karczmarek, Michal Szymczak, and Marcin Paprzycki. Testing the efficiency of jade agent platform. In *ISPDC '04: Proceedings of the Third International Symposium on Parallel and Distributed Computing/Third International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks (ISPDC/HeteroPar'04)*, pages 49–56, Washington, DC, USA, 2004. IEEE Computer Society.

[16] J. Dodgson, M. Spackman, A. Pearman, and L. Phillips. *DTLR multi-criteria analysis manual.* UK: National Economic Research Associates, 2001.

List of Figures

Listings

List of Acronyms

ACL	Agent Communication Language
AUML	Agent Unified Modeling Language
CIC	Client Information Centre
CICIA	Client Information Centre Internal Agent
CIM	Common Information Model
DF	Directory Facilitator
FIPA	Foundation for Intelligent Physical Agents
JADE	Java Agent DEvelopment Framework
FIPASL	FIPA Semantic Language
LMaster	Local Master Agent
LSlave	Local Slave Agent
MCA	Multi Criteria Analysis
OWL	Web Ontology Language
RDF	Resource Description Framework
SPARQL	Protocol And RDF Query Language
QA	Querying Agent
UA	User Agent

Appendix A

Grid Yellow Pages Ontology

```
# Base: http://gridagents.sourceforge.net/YellowPages#
@prefix yellowPages: <http://gridagents.sourceforge.net/YellowPages#> .
@prefix rdfs:     <http://www.w3.org/2000/01/rdf-schema#> .
@prefix rdf:      <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix owl:      <http://www.w3.org/2002/07/owl#> .
@prefix xsd:      <http://www.w3.org/2001/XMLSchema#> .

<http://gridagents.sourceforge.net/YellowPages> a owl:Ontology .

######################
# Team Advertisement #
######################

yellowPages:TeamAd a owl:Class ;
  rdfs:comment "Team advertisement."^^xsd:string ;
  rdfs:subClassOf owl:Thing ;
  rdfs:subClassOf [
    a owl:Restriction ;
    owl:cardinality "1"^^xsd:int ;
    owl:onProperty yellowPages:teamMaster
  ] .

yellowPages:teamMaster a owl:ObjectProperty , owl:FunctionalProperty ;
  rdfs:domain yellowPages:TeamAd ;
  rdfs:range yellowPages:AID .

yellowPages:offersResource a owl:ObjectProperty ;
  rdfs:domain yellowPages:TeamAd ;
  rdfs:range yellowPages:GridResource .

yellowPages:AID a owl:Class ;
  rdfs:comment "Agent IDentifier."^^xsd:string ;
  rdfs:subClassOf owl:Thing ;
  rdfs:subClassOf [
    a owl:Restriction ;
    owl:cardinality "1"^^xsd:int ;
    owl:onProperty yellowPages:aidStr
  ] .

yellowPages:aidStr a owl:DatatypeProperty , owl:FunctionalProperty ;
  rdfs:domain yellowPages:AID ;
  rdfs:range xsd:string .
```

```
##################
# Grid Resources #
##################

yellowPages:GridResource a owl:Class .

yellowPages:ComputerSystem a owl:Class ;
  rdfs:subClassOf yellowPages:GridResource .

yellowPages:ComputerSystemConcept a owl:Class .

yellowPages:Cluster a owl:Class ;
  rdfs:subClassOf yellowPages:ComputerSystem .

yellowPages:UnitaryComputer a owl:Class ;
  rdfs:subClassOf yellowPages:ComputerSystem ;
  rdfs:subClassOf [
    a owl:Restriction ;
    owl:cardinality "1"^^xsd:int ;
    owl:onProperty yellowPages:totalCPUCount
  ] .

yellowPages:totalCPUCount a owl:DatatypeProperty ;
  rdfs:domain yellowPages:UnitaryComputer ;
  rdfs:range xsd:int .

yellowPages:cpu a owl:ObjectProperty ;
  rdfs:domain yellowPages:UnitaryComputer ;
  rdfs:range yellowPages:CPU .

yellowPages:runningOS a owl:ObjectProperty ;
  rdfs:domain yellowPages:UnitaryComputer ;
  rdfs:range yellowPages:OperatingSystemInfo .

yellowPages:hostedFileSystem a owl:ObjectProperty ;
  rdfs:domain yellowPages:UnitaryComputer ;
  rdfs:range yellowPages:FileSystem .

yellowPages:FileSystem a owl:Class ;
  rdfs:subClassOf yellowPages:ComputerSystemConcept .

yellowPages:availableSpace a owl:FunctionalProperty , owl:DatatypeProperty ;
  rdfs:comment "in MB"^^xsd:string ;
  rdfs:domain yellowPages:FileSystem ;
  rdfs:range xsd:int .

yellowPages:CPU a owl:Class ;
  rdfs:subClassOf yellowPages:ComputerSystemConcept ;
  rdfs:subClassOf [
    a owl:Restriction ;
    owl:cardinality "1"^^xsd:int ;
    owl:onProperty yellowPages:cpuCount
  ] .

yellowPages:cpuVendorName a owl:DatatypeProperty ;
  rdfs:domain yellowPages:CPUVendor ;
  rdfs:range xsd:string .

yellowPages:cpuClockSpeed a owl:DatatypeProperty , owl:FunctionalProperty ;
  rdfs:comment "in MHz"^^xsd:string ;
  rdfs:domain yellowPages:CPU ;
  rdfs:range xsd:int .

yellowPages:cpuCount a owl:DatatypeProperty ;
  rdfs:domain yellowPages:CPU ;
  rdfs:range xsd:int .

yellowPages:cpuVendor a owl:ObjectProperty , owl:FunctionalProperty ;
  rdfs:domain yellowPages:CPU ;
  rdfs:range yellowPages:CPUVendor .

yellowPages:cpuArch a owl:ObjectProperty , owl:FunctionalProperty ;
  rdfs:domain yellowPages:CPU ;
  rdfs:range yellowPages:CPUArchictecture .

yellowPages:CPUVendor a owl:Class ;
  rdfs:subClassOf yellowPages:ComputerSystemConcept ;
  rdfs:subClassOf [
    a owl:Restriction ;
    owl:cardinality "1"^^xsd:int ;
    owl:onProperty yellowPages:cpuVendorName
  ] .

yellowPages:CPUArchictecture a owl:Class ;
  rdfs:subClassOf yellowPages:ComputerSystemConcept ;
  rdfs:subClassOf [
    a owl:Restriction ;
    owl:cardinality "1"^^xsd:int ;
    owl:onProperty yellowPages:cpuArchName
  ] .

yellowPages:cpuArchName a owl:DatatypeProperty ;
  rdfs:domain yellowPages:CPUArchictecture ;
  rdfs:range xsd:string .

yellowPages:X86 a owl:Class ;
  rdfs:subClassOf yellowPages:CPUArchictecture .

yellowPages:X86_64 a owl:Class ;
  rdfs:subClassOf yellowPages:X86 .

yellowPages:OperatingSystemInfo a owl:Class ;
  rdfs:subClassOf yellowPages:ComputerSystemConcept ;
  rdfs:subClassOf [
    a owl:Restriction ;
    owl:cardinality "1"^^xsd:int ;
    owl:onProperty yellowPages:operatingSystem
  ] .

yellowPages:freePhysicalMemory
  a owl:FunctionalProperty , owl:DatatypeProperty ;
  rdfs:comment "in MB"^^xsd:string ;
  rdfs:domain yellowPages:OperatingSystemInfo ;
  rdfs:range xsd:int .

yellowPages:freeVirtualMemory
  a owl:DatatypeProperty , owl:FunctionalProperty ;
  rdfs:comment "in MB"^^xsd:string ;
  rdfs:domain yellowPages:OperatingSystemInfo ;
  rdfs:range xsd:int .

yellowPages:installedSoftware a owl:ObjectProperty ;
  rdfs:domain yellowPages:OperatingSystemInfo ;
  rdfs:range yellowPages:Software .

yellowPages:operatingSystem a owl:ObjectProperty ;
  rdfs:domain yellowPages:OperatingSystemInfo ;
  rdfs:range yellowPages:OperatingSystem .

yellowPages:OperatingSystem a owl:Class ;
  rdfs:subClassOf yellowPages:ComputerSystemConcept .

yellowPages:osType a owl:FunctionalProperty , owl:DatatypeProperty ;
  rdfs:domain yellowPages:OperatingSystem ;
  rdfs:range xsd:string .

yellowPages:osVersion a owl:FunctionalProperty , owl:DatatypeProperty ;
  rdfs:domain yellowPages:OperatingSystem ;
  rdfs:range xsd:string .

yellowPages:Windows a owl:Class ;
  rdfs:subClassOf yellowPages:OperatingSystem .

yellowPages:Unix a owl:Class ;
  rdfs:subClassOf yellowPages:OperatingSystem .

yellowPages:Linux a owl:Class ;
  rdfs:subClassOf yellowPages:Unix .

yellowPages:Software a owl:Class ;
  rdfs:subClassOf yellowPages:ComputerSystemConcept ;
  rdfs:subClassOf [
    a owl:Restriction ;
    owl:cardinality "1"^^xsd:int ;
    owl:onProperty yellowPages:softwareName
  ] .

yellowPages:softwareName a owl:DatatypeProperty ;
  rdfs:domain yellowPages:Software ;
  rdfs:range xsd:string .

yellowPages:softwareVersion
  a owl:FunctionalProperty , owl:DatatypeProperty ;
  rdfs:domain yellowPages:Software ;
  rdfs:range xsd:string .

##################################
# Instances of CPU Architectures #
##################################

yellowPages:Generic_x86 a yellowPages:X86 ;
  yellowPages:cpuArchName "x86"^^xsd:string .

yellowPages:Alpha a yellowPages:CPUArchictecture ;
  yellowPages:cpuArchName "Apha"^^xsd:string .

yellowPages:PowerPC a yellowPages:CPUArchictecture ;
  yellowPages:cpuArchName "PowerPC"^^xsd:string .

yellowPages:SPARC a yellowPages:CPUArchictecture ;
  yellowPages:cpuArchName "SPARC"^^xsd:string .

yellowPages:PA_RISC a yellowPages:CPUArchictecture ;
  yellowPages:cpuArchName "PA-RISC"^^xsd:string .
```

```
yellowPages:MIPS a yellowPages:CPUArchictecture ;
  yellowPages:cpuArchName "MIPS"^^xsd:string .

yellowPages:IA_64 a yellowPages:CPUArchictecture ;
  yellowPages:cpuArchName "IA-64"^^xsd:string .

yellowPages:EMT64 a yellowPages:X86_64 ;
  yellowPages:cpuArchName "EMT64"^^xsd:string .

yellowPages:AMD64 a yellowPages:X86_64 ;
  yellowPages:cpuArchName "AMD64"^^xsd:string .

############################
# Instances of CPU Vendors #
############################

yellowPages:Intel a yellowPages:CPUVendor ;
  yellowPages:cpuVendorName
  "Intel"^^xsd:string .

yellowPages:IBM a yellowPages:CPUVendor ;
  yellowPages:cpuVendorName
  "IBM"^^xsd:string .

yellowPages:AMD a yellowPages:CPUVendor ;
  yellowPages:cpuVendorName "AMD"^^xsd:string .
```

Appendix B

Messaging Ontology

```
# Base: http://gridagents.sourceforge.net/Messaging#
@prefix xsd:     <http://www.w3.org/2001/XMLSchema#> .
@prefix msg:  <http://gridagents.sourceforge.net/Messaging#> .
@prefix jade:    <http://jade.cselt.it/beangenerator#> .
@prefix rdfs:    <http://www.w3.org/2000/01/rdf-schema#> .
@prefix rdf:     <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix owl:     <http://www.w3.org/2002/07/owl#> .

<http://gridagents.sourceforge.net/Messaging> a owl:Ontology ;
      owl:imports <http://jade.cselt.it/beangenerator> .

##################
# Agent Actions #
##################

msg:CICAction a jade:JADE-CLASS ;
  rdfs:comment "Base class of actions handled by CIC agent."^^xsd:string ;
  rdfs:subClassOf jade:AgentAction .

msg:CICQueryResource a jade:JADE-CLASS ;
  rdfs:comment "Action of querying for resources in yellow pages."
    ^^xsd:string ;
  rdfs:subClassOf msg:CICAction .

msg:resourceQuery a owl:ObjectProperty , owl:FunctionalProperty ;
  rdfs:domain msg:CICQueryResource ;
  rdfs:range msg:OntoQuery .

msg:CICTeamAction a jade:JADE-CLASS ;
  rdfs:comment "Base class for team related actions on yellow pages.
    Provides authentication data."^^xsd:string ;
  rdfs:subClassOf msg:CICAction .

msg:teamURI a owl:FunctionalProperty , owl:DatatypeProperty ;
  rdfs:domain msg:CICTeamAction ;
  rdfs:range xsd:string .

msg:teamPassword a owl:DatatypeProperty , owl:FunctionalProperty ;
  rdfs:domain msg:CICTeamAction ;
  rdfs:range xsd:string .

msg:CICAddTeam a jade:JADE-CLASS ;
```

```
  rdfs:comment "Action of adding team advertisement to yellow pages."
    ^^xsd:string ;
  rdfs:subClassOf msg:CICTeamAction .

msg:teamAd a owl:ObjectProperty , owl:FunctionalProperty ;
  rdfs:comment "Team advertisement."^^xsd:string ;
  rdfs:domain msg:CICAddTeam ;
  rdfs:range msg:OntoData .

msg:CICUpdateTeam a jade:JADE-CLASS ;
  rdfs:comment "Action of updating team advertisement in yellow pages."
    ^^xsd:string ;
  rdfs:subClassOf msg:CICTeamAction .

msg:CICRemoveTeam a jade:JADE-CLASS ;
  rdfs:comment "Action of removing team advertisement from yellow pages."
    ^^xsd:string ;
  rdfs:subClassOf msg:CICTeamAction .

msg:CICAddResource a jade:JADE-CLASS ;
  rdfs:comment "Action of adding resource to team advertisement
    in yellow pages."^^xsd:string ;
  rdfs:subClassOf msg:CICUpdateTeam .

msg:resourceAd a owl:ObjectProperty , owl:FunctionalProperty ;
  rdfs:comment "Resource advertisement."^^xsd:string ;
  rdfs:domain msg:CICAddResource ;
  rdfs:range msg:OntoData .

msg:CICRemoveResource a jade:JADE-CLASS ;
  rdfs:comment "Action of removing a resource from team advertisement
    in yellow pages."^^xsd:string ;
  rdfs:subClassOf msg:CICUpdateTeam .

msg:resourceAdURI a owl:DatatypeProperty , owl:FunctionalProperty ;
  rdfs:comment "URI of resource advertisement."^^xsd:string ;
  rdfs:domain msg:CICRemoveResource ;
  rdfs:range xsd:string .

##################
# Action Results #
##################

msg:ActionResult a jade:JADE-CLASS ;
  rdfs:comment "Base class for all action results."^^xsd:string ;
  rdfs:subClassOf jade:Concept .

msg:OntoQueryResult a jade:JADE-CLASS ;
  rdfs:comment "Base class for result of ontology query."^^xsd:string ;
  rdfs:subClassOf msg:ActionResult .

msg:SelectQueryResult a jade:JADE-CLASS ;
  rdfs:comment "ResultSet of SELECT query."^^xsd:string ;
  rdfs:subClassOf msg:OntoQueryResult .

msg:selectQueryResult a owl:ObjectProperty , owl:FunctionalProperty ;
  rdfs:domain msg:SelectQueryResult ;
  rdfs:range msg:OntoData .

msg:AskQueryResult a jade:JADE-CLASS ;
  rdfs:comment "Boolean result of ASK query."^^xsd:string ;
  rdfs:subClassOf msg:OntoQueryResult .
```

```
msg:askQueryResult a owl:FunctionalProperty , owl:DatatypeProperty ;
  rdfs:domain msg:AskQueryResult ;
  rdfs:range xsd:boolean .

msg:DescribeQueryResult a jade:JADE-CLASS ;
  rdfs:comment "RDF graph result of DESCRIBE query."^^xsd:string ;
  rdfs:subClassOf msg:OntoQueryResult .

msg:describeQueryResult a owl:ObjectProperty , owl:FunctionalProperty ;
  rdfs:domain msg:DescribeQueryResult ;
  rdfs:range msg:OntoData .

msg:ConstructQueryResult a jade:JADE-CLASS ;
  rdfs:comment "RDF graph result of CONSTRUCT query."^^xsd:string ;
  rdfs:subClassOf msg:OntoQueryResult .

msg:constructQueryResult a owl:ObjectProperty , owl:FunctionalProperty ;
  rdfs:domain msg:ConstructQueryResult ;
  rdfs:range msg:OntoData .

msg:ActionFailure a jade:JADE-CLASS ;
  rdfs:comment "Base class for all action failures."^^xsd:string ;
  rdfs:subClassOf msg:ActionResult .

msg:failureMsg a owl:DatatypeProperty , owl:FunctionalProperty ;
  rdfs:comment "Message describing type of failure."^^xsd:string ;
  rdfs:domain msg:ActionFailure ;
  rdfs:range xsd:string .

msg:CICFailure a jade:JADE-CLASS ;
  rdfs:comment "Base class for all CIC action failures."^^xsd:string ;
  rdfs:subClassOf msg:ActionFailure .

msg:CICInternalFailure a jade:JADE-CLASS ;
  rdfs:comment "Internal CIC failure."^^xsd:string ;
  rdfs:subClassOf msg:CICFailure .

msg:TeamDoesNotExist a jade:JADE-CLASS ;
  rdfs:comment "Specified team does not exist."^^xsd:string ;
  rdfs:subClassOf msg:CICFailure ;
  jade:JADE-JAVA-CODE "public void test() {}"^^xsd:string .

msg:InvalidTeamPassword a jade:JADE-CLASS ;
  rdfs:comment "Authentication failed because of invalid team password."
    ^^xsd:string ;
  rdfs:subClassOf msg:CICFailure .

msg:TeamAlreadyExists a jade:JADE-CLASS ;
  rdfs:subClassOf msg:CICFailure .

############
# Concepts #
############

msg:OntoQuery a jade:JADE-CLASS ;
  rdfs:comment "Serialized ontology query like SPARQL or RDQL.
    See oneOf for complete list."^^xsd:string ;
  rdfs:subClassOf jade:Concept .

msg:ontoQueryStr a owl:FunctionalProperty , owl:DatatypeProperty ;
  rdfs:comment "Serialized ontology query."^^xsd:string ;
  rdfs:domain msg:OntoQuery ;
  rdfs:range xsd:string .

msg:ontoQuerySyntax a owl:FunctionalProperty , owl:DatatypeProperty ;
  rdfs:domain msg:OntoQuery ;
  rdfs:range [
    a owl:DataRange ;
    owl:oneOf ("SPARQL"^^xsd:string "RDQL"^^xsd:string)
  ] .

msg:OntoData a jade:JADE-CLASS ;
  rdfs:comment "Serialized ontology data like RDF/XML or RDF/XML-ABBREV.
    See oneOf for complete list."^^xsd:string ;
  rdfs:subClassOf jade:Concept .

msg:ontoDataStr a owl:DatatypeProperty , owl:FunctionalProperty ;
  rdfs:comment "Serialized ontology data."^^xsd:string ;
  rdfs:domain msg:OntoData ;
  rdfs:range xsd:string .

msg:baseURI a jade:JADE-SLOT , owl:DatatypeProperty ,
    owl:FunctionalProperty ;
  rdfs:domain [
    a owl:Class ;
    owl:unionOf (msg:OntoData msg:OntoQuery)
  ] ;
  rdfs:range xsd:string ;
  jade:JADE-NAME "myBaseURI"^^xsd:string ;
  jade:JADE-UNNAMED-SLOT "true"^^xsd:boolean .

msg:ontoLang a owl:FunctionalProperty , owl:DatatypeProperty ;
  rdfs:comment "Serialization language."^^xsd:string ;
  rdfs:domain msg:OntoData ;
  rdfs:range [
    a owl:DataRange ;
    owl:oneOf ("RDF/XML"^^xsd:string "RDF/XML-ABBREV"^^xsd:string
      "N3"^^xsd:string "N-TRIPLE"^^xsd:string "TURTLE"^^xsd:string)
  ] .
```

Appendix C

CIC behaviours

Here we present Java code snippets of behaviors that are part of *CICIA* and *CIC Agent*. All behaviors are defined as static inner classes.

```
/**
  * Receives client requests from jade message queue and
  * puts them in local request queue for processing taking
  * into account priority of the request.
  */
private class RequestsListener extends CyclicBehaviour {

  public void action() {
    ACLMessage msg = myAgent.receive(getRequestMsgTemplate());
    if (msg != null) {
      try {
        requestQueue.put(msg);
      } catch (QueueLimitException e) {
        log.warning("Request␣queue␣limit␣reached.");
      }
    }
    else {
      block();
    }
  }

} // class RequestListener


/**
* Delegates client requests to free db agents for execution.
*/
static class RequestsDelegator extends CyclicBehaviour {

  private MyQueue requestQueue;
  private MyQueue releasedDbAgents;
  private AgentPool dbAgents;
  private AID resultsReceiver;

  /**
  * Associates this behavior with requestQueue and releasedDbAgents queues.
  * @param ap AgentPool from where db agents are to be taken.
  * @param releasedDbAgents Db agents that recently returned results.
  * @param requestQueue RequestQueue to be read from.
  * @param resultsReceiver Agent which is going to process results
  * from db agents.
  */
  public RequestsDelegator(AgentPool ap, MyQueue releasedDbAgents,
      MyQueue requestQueue, AID resultsReceiver) {
    this.dbAgents = ap;
    this.releasedDbAgents = releasedDbAgents;
    this.requestQueue = requestQueue;
    this.resultsReceiver = resultsReceiver;

    // be notified about new requests to be handled
    requestQueue.associate(this);
    // be notified about db agents recently returning results
    releasedDbAgents.associate(this);
  }

  public void action() {
    // check first if we have available db agents
    if (releasedDbAgents.size() > 0 || dbAgents.getFreePoolSize() > 0) {
      ACLMessage msg = (ACLMessage)requestQueue.get();
      if (msg != null) {
        // check first if db agent has been recently released
        AID db = (AID)releasedDbAgents.get();
        if (db == null) {
          // acquire new lock
          db = dbAgents.acquireAgent();
        }

        // forward request to db agent
        prepareDbRequestMsg(msg, myAgent.getAID(), resultsReceiver, db);
        myAgent.send(msg);
      }
      else {
        // nothing to do, release locks if any
        AID db = (AID)releasedDbAgents.get();
        if (db != null) {
          dbAgents.releaseAgent(db);
        }
        else {
          // no recently released agents, nothing to do
          block();
        }
      } // if (msg != null)
    }
    else {
      // no free db agents
      block();
    } // if (releasedDbAgents.size() > 0 || dbAgents.getFreePoolSize() > 0)
  }

} // class RequestsDelegator


/**
  * Listens for incoming results from db agents
  * and puts them in result queue.
  */
static class DbResultsListener extends CyclicBehaviour {

  private Logger myLog;
  private MyQueue resultQueue;
  public DbResultsListener(MyQueue resultQueue) {
    this.resultQueue = resultQueue;
  }

  public void onStart() {
    myLog = Logger.getMyLogger(myAgent.getClass().getName());
    myLog.config("Listening␣for␣db␣results...");
  }

  public void action() {
    ACLMessage msg = (ACLMessage)myAgent.receive(getReplyMsgTemplate());
    if (msg != null) {
      try {
        resultQueue.put(msg);
      } catch (QueueLimitException e) {
        myLog.severe("Should␣never␣happen!");
      }
    }
    else {
      block();
    }
  }

} // DbResultsListener


/**
  * Reads result from result queue, releases db agent, sends result
  * to client.
  */
static class ResultsSender extends CyclicBehaviour {

  private Logger myLog = Logger.getMyLogger(CICAgent.class.getName());
  private MyQueue resultQueue;
  private ResultHandler resultHandler;
  private MyQueue releasedDbAgents;

  public ResultsSender(MyQueue resultQueue, ResultHandler resultHandler,
      MyQueue releasedDbAgents) {
    this.resultQueue = resultQueue;
    this.resultHandler = resultHandler;
    this.releasedDbAgents = releasedDbAgents;

    // be notified about new results to be sent
    resultQueue.associate(this);
  }

  public void action() {
    ACLMessage msg = (ACLMessage)resultQueue.get();
    if (msg != null) {
      try {
        // release db agent
        releasedDbAgents.put(msg.getSender());
      } catch (QueueLimitException e) {
        myLog.severe("This␣should␣never␣happen!");
      }
      // send result
      resultHandler.handleResult(msg);
    }
    else {
      block();
    }
  }
} // class ResultsSender
```

Warszawa, 27 Październik 2006

Oświadczenie

Oświadczam, że pracę magisterską pod tytułem "Intelligent Software Agents in Resource Management on the Grid", której promotorem jest Prof. Marcin Paprzycki wykonałem samodzielnie, co poświadczam własnoręcznym podpisem.

....