MODELING VOLUNTEER COMPUTING GRID ON THE WEB BY SOCIALLY INTELLIGENT AGENTS

Wei Li
Faculty of Informatics and Communication
Central Queensland University
North Rockhampton, QLD 4702, Australia

Xinghuo Yu
Faculty of Engineering
RMIT University
Melbourne, VIC 3001, Australia

Abstract

This paper presents our first-step efforts towards agent-based computing grid on the Web. When there are several prototypes addressing volunteer computing, our model is actually the one that focuses on open issues in them, i.e., Adaptive Parallelism, Scalability, Applicability and Fault-Tolerance. The motivation of our work is from the growing consensus that socially intelligent agents naturally model complex systems in an uncertain and changing environment. For being applicable to the real Web environment, in the design of our model, we do not limit the number of volunteers or clients; we do not require volunteers’ persistence; we support fine-grained applications and tolerate potential crash of volunteer machines. Based on ideas of this paper, we have set up a project, and the first version of the prototype system, called Supagents 1.0, has been developed. So far, some of the claimed features of our model have been demonstrated by dedicatedly designed benchmark applications.

Keywords: Volunteer Computing, BDI agents, Multi-Agent Systems, Metacomputing, Computing Grid.

1. Introduction

This paper presents the idea of agent-based volunteering computing on the Web, which aims at high-performance parallel computing networks to be formed easily, quickly and inexpensively by enabling ordinary Web users to harness their computers’ idle processing power without needing professional skills for setting up. A good motivation for volunteer computing could be the conventional belief that the Web is connecting millions of mostly idle machines. Harnessing these idle machines could prove to be a powerful global computing infrastructure. Volunteer computing makes it easy for people on the Web to pool their existing computer resources, and work together to solve large computational problems that no one can solve alone. Volunteer Computing, also called Global Computing, Metacomputing or Supercomputing, makes it possible to build parallel and distributed computing networks much larger and much more quickly than possible before by simply browsing a web page. Potentially, such networks can involve thousands, even millions of computers distributed around the world, and can achieve performance levels of current supercomputers.

The first appearance of the idea of volunteer computing [14] was in 1996. Since then several projects, such as Bayanihan [15], Javelin [6], ATLAS [1], Charlotte [2] and Amica [8] etc., have been set up. So far, those projects have produced prototypes. However, several important issues or challenges have not been investigated or well investigated or well modeled by these prototypes. These open issues are as follows.

1. Adaptive Parallelism: dynamically exploit a varying collection of computational resources, so that the running tasks can be migratable, and the computational persistence and parallelism for an application can be maintained. The highly heterogeneous and dynamic natures of the Web challenge volunteer computing systems. Volunteer computers can join and leave computing at any time. The volunteer computing systems must be modeled adaptively parallel. They must not assure the existence of a fixed number of volunteers, or depend on any static timing information about the system.

2. Scalability: dynamically scale to a huge number of computations, which may involve millions of volunteers and clients. Web-based volunteer computing is highly security-restricted. Programs, usually Java applets as clients or volunteers running in browsers, can only communicate with the Web server from which they were downloaded. This forces Web-based volunteer systems into a simple star topology, which has the demerits of having high congestions, no parallelism and not being scalable. To maximize the benefits of volunteer computing, it would be useful to develop techniques for better scalability.

3. Applicability: allow programmers to implement a wide variety of parallel applications easily and quickly by general-purpose and easy-to-use programming interfaces. Although coarse-grained applications are ideal for volunteer computing systems because they work well even with slow internet links, and can easily be made adaptively parallel, the volunteer computing system need not to be limited to coarse-grained applications, as current belief and practice may suggest, but can be used for other grained applications that may use more complex communication patterns.

A growingly large numbers of Java-based systems have been developed for addressing computing grid since the first release of Java. Among these systems are ATLAS [1], ParaWeb [4], JPVM [7], and Manta [9]. Being application-based systems, they do not have the properties of easy-to-use and accessibility as applet-based systems. There are a few projects that have implemented web-based volunteer computing by using applets. More recent ones, including
Charlotte [2], Javelin [6], JET [10], and Bayanihan [14, 15], are much easier to use than application-based systems since they allow a volunteer to join the volunteer system by simply visiting a web page. However, because of applet security restrictions, the need for adaptive parallelism and crash-tolerance seems to have so far limited most of these applet-based systems. So far, up to our best knowledge, we have not found volunteer systems by using Multi-agent technology or BDI agents, which will be demonstrated to have great advantages.

2. BDI Agents

Intelligent Agent is a powerful Artificial Intelligence technology, which shows considerable promise as a new paradigm for mainstream software development. There is a growing consensus that agent systems have at least the properties: proactiveness, reactivity, autonomy and being situated [16]. An intelligent agent is able to make rational decisions, i.e., blending proactiveness and reactivity, showing rational commitment to decisions made, and exhibiting flexibility in the face of an uncertain and changing environment [11]. As a result, it is natural for us to use multi-agents to model a scalable, adaptive and general-purpose infrastructure for volunteer computing, which is able to be applicable to the Web, a highly and dynamic environment.

Among other models, the BDI model [12, 13] comes from research done in the field of artificial intelligence over the past twenty years, and has proven to be the most robust and flexible model for Intelligent Agent Systems. It has its basis in philosophy [3], and offers a logical theory which defines the mental attitudes of Belief, Desire and Intention using a modal logic [18]. The central concepts in the BDI model are as follows.

1. Beliefs: represent the agent’s current knowledge about the world, including information about the current state of the environment inferred from perception devices and message from other agents, as well as internal states.
2. Desires: represent a motivational state of the system, which the agent is trying to achieve.
3. Intentions: represent the chosen means to achieve the agent’s desires, and are generally implemented as plans.

These mental attitudes determine the system’s behavior, and are critical for achieving adequate or optimal performance when deliberation is subject to resource bounds. Our experience with intelligent agents leads us to the firm conviction that the agent-oriented approach is particularly useful for building complex distributed systems involving resource-bounded decision-making. Our model aims at supporting true volunteer computing, which means that their participants are volunteers in the true sense of the word in that they: come and leave of their own free will and are unknown to the administrators before they volunteer, i.e., they can come from anywhere, not just from the administrators’ domains. Our model will exhibit more flexibility when addressing the open issues.

3. Topology of Computing Grid by a Multi-Agent System

In our model, there are four participating entities or roles (agents): Client, Worker, Manager and General-Manager. A client is seeking computing resources, and a worker is offering computing resources. A manager coordinates the supply and demand for computing resources. With these 3 roles, the simple star network is straightforward to set up and is good for relatively small number of clients or workers. Also, the adoption of this simple star topology is forced by the security restrictions from browser-based networks of the Web. However, there is only one manager in a simple star network, thus, as the number of clients or workers increase, the server (manager) becomes a communication and computational bottleneck because all communication must be routed through the server.

![Simple star and adaptive star topologies](image)

In a network of managers, a single server acts as a General Manager (GM). This server is responsible for holding the entry home page, taking care of redirecting clients and volunteers to managers within the network. When a client joins the network by accessing the URL published at the GM (as indicated by the dashed arrow between C1 and the GM), the join will be redirected to an appropriate server (the least work-loaded manager). When the client has connected to the appropriate server, a final socket connection is made (as indicated by the solid arrow between C1 and M3). A worker has the same join progress except for being redirected to the most work-loaded server. We have noted that the GM could be a dedicated server or one of managers. Nothing changes from the clients’ point of view, but now the server becomes a network composed of several server computers, each capable of acting as a manager of accepting application graphs, distributing computations, routing inter-computation communications, managing workers’ random leave and collecting and reporting computational results (detailed in next section). A key assumption in this topology is that all managers are geographically distributed so that the communication bottleneck could be really avoided.

4. Application Graph

4.1 Application Graph

Firstly, we are aware of the importance of representing an application in the form of Application Graph. An application graph is constructed using components representing requests to the infrastructure for complex computational and storage services, as well as application-relevant functionalities and data-dependences among components. An application graph is a set consisting of
Computations in an ordered set C and Data-Dependences in a set D, i.e., an application graph AG=(C, D).

A computation is a computational entity with an application-relevant functionality, i.e., encapsulating behavior and data. It can have arbitrary number input parameters and output results, which take the form of a vector of objects. A data-dependence represents a data-transfer between two computations, meaning that the start of one computation depends on the completion of other computations. A data-dependence requires inter-computation communication or inter-machine communication if computations are on different machines.

These abstract computations and data-dependences must be translated into concrete data-flow or control-flow graph, by which the infrastructure can manage computation distributions, inter-computation communication and result collections. An application graph will be initialized into an Application Execution Graph (AEG), which will be dynamically maintained during it execution. Firstly, we form an Order, the execution or distribution order of computations, from an application graph. Secondly, we have to figure out and represent Data-Dependences among computations. During the running time of an application graph, computations are dynamically distributed to available volunteer machines (modeled as Worker agents). We don’t need any limit to the number of computations and to the number of workers to run an application graph, i.e., there is at least one worker for an application graph to be run even if the performance could be very low in this case. In addition, an AEG is also a central cache of computational results of computations from workers. In the Table 1, static information of an application graph, i.e., Computations, their Order or Data-Dependences, is highly application-relevant, and it should be defined by application programmers. However, dynamic information, i.e., Distribution and Results, is dynamically created and maintained by a manager at runtime.

<table>
<thead>
<tr>
<th>Computation</th>
<th>Order</th>
<th>Data-Dependence</th>
<th>Distribution</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>0</td>
<td>In[c5]</td>
<td>work1, work5</td>
<td>null</td>
</tr>
<tr>
<td>c0</td>
<td>1</td>
<td>In[c2, c3]</td>
<td>work2, work8</td>
<td>null</td>
</tr>
<tr>
<td>c3</td>
<td>2</td>
<td>In[c4, c5]</td>
<td>work3</td>
<td>null</td>
</tr>
<tr>
<td>c5</td>
<td>3</td>
<td>In[c1, c3]</td>
<td>work4</td>
<td>null</td>
</tr>
<tr>
<td>c2</td>
<td>4</td>
<td>In[c0]</td>
<td>work5</td>
<td>null</td>
</tr>
<tr>
<td>c4</td>
<td>5</td>
<td>In[c2, c3]</td>
<td>work6</td>
<td>null</td>
</tr>
</tbody>
</table>

Table 1: An example AEG of the AG in Figure 2

4.2 Computation

A computation is a basic unit for computing as well as for scheduling. That means a computation is a migratable object between a manager and its workers. Thus, a computation could be defined as follows.

```
public class Computation implements Serializable {
    private Vector parameters;
    private int numOfParameters;
    public void addParameters(Vector ve)
    {merge ve with parameters into parameters;}
    public boolean checkParameterNumber()
    {if (parameters.size()>=numOfParameters) return true else return false;}
    public Vector execute()
    {extract parameters from the Vector parameters;
    execute functionalities;
    return a Vector of objects as results;}
}
```

A computation could not be executed until it has the required parameters, i.e., its data-dependence has been satisfied. This could be checked by method CheckParameterNumber. The functionalities of a computation are included in the method Execute. And the parameters for a computation are saved in a vector of objects parameters, which could be dynamically accumulated by invoking method addParameters.

A computation lives in the following 5 states, i.e., Abortion, Standby, Ready, Running and Termination, during its lifecycle. When an application graph is initialized into an AEG, all the computations in an AEG are in the state Standby. When the data-dependences for a computation are satisfied, a computation changes its state from Standby to Ready. Once a ready computation is distributed to a worker to execute, it is in the Running state. If a computation’s running successfully finishes, it turns into the Termination state before garbage collection. During any state of Standby, Ready, and Running, a computation may come to the state Abortion due to a client aborts his/her application. A running computation may come back to the Ready state because of the dependent worker’s unpredictable leave or crash. Thus, on behalf of clients, a manager acts as a central cache of AEGs before their successful terminations. Consequently, if we support mobile computing, The GM should act as a central cache for clients’ results when clients are permitted to disconnect from the Web and want to retrieve results later on.

After the computation is well defined, an application graph could be defined by implementing the following interface. From these three methods in the interface, a manager can extract static information of an application graph, i.e., computations, their order or data-dependences, either input or output.

```
public interface applicationGraph implements Serializable {
    public Vector getComputations();
    public Vector getIndependence();
    public Vector getOutdependences();
}
```

4.3 Scheduling of Computations

To a manager, computations are arranged into three queues whatever applications they are from. When a client submits an application graph, computations are extracted from it and put into the Waiting Queue (WQ). If a computation in the WQ has no input data-dependences, or its input data-dependences have been satisfied, it will move into the Ready
Queue (RQ). When a computation is distributed to a worker, it will move into the Execution Queue (EQ). A computation could move back into the RQ because its dependent worker’s leave. The EQ is circular queue with a pointer keeping track of the next available uncompleted (Running) computations for multiple distributions. We make use of Eager Scheduling once the number of remaining computations becomes less than the number of available workers, Eager Scheduling aggressively assigns and re-assigns computations to available workers until all computations have been executed to completion. Concurrent assignments of computations to multiple workers guarantee that slow workers (machines), even very slow workers, do not slow down the progress of an application graph. Furthermore, if workers crash or become less accessible, for example due to network delays, and if at least one worker is available for a sufficiently long period of time, the entire application graph will finish. Note that eager scheduling masks failures without the need to actually detect failures. In fact, failure is a special case of a slow machine (an infinitely slow machine).

5. Agents Modeling of Volunteer Computing

As a tool, we use JACK-like agent language to formally describe our modeling. JACK [5] is a Java-based and commercially available agent language implementing the Belief, Desire and Intention agent paradigm. It supports the use of plans, and JACK agents can be configured with a set of plans to use. Each JACK plan must be designated to handle a class of event. An event can either be something sent between agents (messages), something posted by the agent to be handled within itself (signifying a change in desires/goals or beliefs) or something externally generated from the environment. When deciding which plan to execute for an agent, JACK will select a set of plans designed to handle the incoming events. It will then disregard any plan that has conditions, which mean that it is inapplicable to this particular event. JACK then selects a plan from the remaining set of applicable plans. In order to focus on the description of mainstream behaviors of our modeling, we neglect trivial details in JACK for programming a real JACK agent, and use a JACK-like language instead. We have identified necessary roles for such a complex system, and those roles are modeled as agents. The whole system is triggered running by events, either internal or external to agents. From the agent society’s point of view, an event represents a goal that an agent hopes to achieve.

1. Worker: a worker agent is on behalf of a volunteer computer. It is a computing engine. Its responsibilities include receiving computations or parameters, executing computations, and returning computational results.

2. Client: a client is on behalf of a user. It submits an application graph to a manager, and waits for the computational results.

3. Manager: a manager manages workers, accepting their registrations as volunteer computing nodes, and accepting their deregistrations when they leave the system. A manager manages clients, accepting their submissions of application graphs, and returning computational results to them. The most important responsibility of a manager is to guarantee that application graphs can be successfully executed when volunteer workers are permitted to join and leave the system dynamically, i.e., the computational persistence and adaptive parallelism for an application graph must be maintained once the application graph is initialized for execution.

4. General-Manager: there is only one general manager in the system so that we only need to publish a URL for clients or volunteers to access the system. The responsibility of the general manager is to manage load balancing among managers of the system so that the entire system is scalable to hold a large number of clients and workers. When a client or a worker joins the system, the general manager redirects it to a manager according to the strategy of load balancing.

Because of the length limit to the paper, we only provide the formal descriptions of a manager’s BDI behaviours as follows.

Beliefs: { PW: a list of postal addresses of available workers; PC: a list of postal addresses of clients; The general manager’s postal address; A list of references of AEGs; The references of queues WQ, RQ and EQ; }

Capability: {#use plan planRegistration; //to receive a client’s or a worker’s registration. #use plan planDeregistration; //to receive a client’s or a worker’s deregistration. #use plan planSubmission; //to receive a client’s submission of an application graph. #use plan planDistribution; //to do eager scheduling #use plan planResults; //to receive computational results from a worker. #use plan planAbortion; //to abort a client’s application } plan planRegistration
{#handle external event eventJoin; //from a client or a worker. #send internal event eventDistribution; //to itself. body { if (a client joins) put the client’s postal address into PC else put the worker’s postal address into PW; }
6. Performance Results

The prototype has been implemented by JACK with an extension of supporting the proposed Adaptive Star Topology and applet agents. So far, two typical benchmarks in distributed and parallel computing areas, Brute-Force Factoring and Computing π, have been built up for testing the performance of our initial prototype. Experiments were conducted in campus computer labs under normally daily workload. We had 20 Pentium III 933 computers, and 4 of them were used for dedicated servers and the others were used for volunteers. Clients are any computers in labs except for the above 20s. The computers were connected by 10 Mbit Ethernet network. We deliberately decomposed the benchmark applications into different number and different size computations so that they were made computation-intensive or communication-intensive. In one application, Brute-Force Factoring, we have written an application to find all the factors of the largest value (9223372036854775807) of a Java long integer. In order to increase computational and communication intensities, we submitted 100 largest integers concurrently. In the other application, Computing π, we exploited the Monte Carlo method by randomly generating points within 2×2 square and counting how many points are within a unit circle [17]. Detailed properties of two applications are summarized in Table 2.

### Table 2: Properties of application graphs of Computing π and Brute-Force Factoring applications

<table>
<thead>
<tr>
<th>Application</th>
<th>Computing π (concurrently computing 100 integers)</th>
<th>Brute-Force Factoring (concurrently factoring 100 integers)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computations</td>
<td>480,000 (first run)</td>
<td>303,800 (first run)</td>
</tr>
<tr>
<td>/per computation</td>
<td>240,000,000 (second run)</td>
<td>151,850,000 (second run)</td>
</tr>
<tr>
<td>Iterations</td>
<td>1 million (first run)</td>
<td>1 million (first run)</td>
</tr>
<tr>
<td>/per volunteer</td>
<td>2,000 (second run)</td>
<td>2,000 (second run)</td>
</tr>
<tr>
<td>Computations-intensity</td>
<td>30,000 (first run)</td>
<td>18,987 (first run)</td>
</tr>
<tr>
<td>Communication-intensity</td>
<td>15,000,000 (second run)</td>
<td>9,490,625 (second run)</td>
</tr>
</tbody>
</table>

In order to check the adaptive parallelism and scalability of our model, we did two experiments. In one experiment, we configured our volunteer network into a simple star topology with a general-manager, a manager and maximum 16 volunteers to simulate the topology of other state-of-art volunteer systems. In the other experiment, the volunteer network was configured into an adaptive star topology with a general-manager, three managers and maximum 16 volunteers. The performance results for the Brute-Force Factoring and Computing π applications were similar. We report the speedup curves of the Brute-Force Factoring within different network topologies in the following figures.
7. Conclusion and Future Work

When there are several systems for volunteer computing, our model is actually the one that focuses on open issues in current volunteer computing systems. We aim at improving adaptive parallelism, scalability, applicability and fault-tolerance, which determine if a volunteer system is able to be applicable to the Web environment. In our model, adaptive parallelism has been implemented by writing managers that follow Eager Scheduling strategy. In this way, we get a simple form of locally dynamic load-balancing and fault-tolerance. Our mechanism of manager network is in such a way to dynamically redirect volunteers or clients to appropriate servers for global load-balancing and scalability as well. Our model need not to be limited to master-worker style applications or coarse-grained applications, but can also be used for message-passing style applications. Our model is general-purpose, and allows programmers to write a variety of applications by simply implementing Application Graphs. For testing the fault-tolerant performance, we intentionally crashed some workers by simply closing their browser windows. Figure 7 shows that the performance decreased when crashes happened. However, when the workers rejoined or restored into the system, performance re-increased again.

8. References

[14] Luis F.G. Sarmenta, Volunteering Computing, a preliminary concept and project proposal, MIT Laboratory for Computer Science, October 1996
[16] John Thangarajah, Representation and Reasoning for Goals in BDI Agents, Australian Computer Science Conference, Jan 2002