Future Generation Computer Systems 28 (2012) 854–860

Contents lists available at SciVerse ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Decentralized approach to resource availability prediction using group availability in a P2P desktop grid

Karthick Ramachandran *, Hanan Lutfiyya, Mark Perry

Department of Computer Science, University of Western Ontario, London, Ontario, Canada

ARTICLE INFO

Article history:
Received 15 June 2010
Accepted 12 October 2010
Available online 18 November 2010

Keywords:
Peer-to-peer desktop grids
Resource availability
Cloud computing
Group availability

ABSTRACT

In a desktop grid model, the job (computational task) is submitted for execution in the resource only when the resource is idle. There is no guarantee that the job which has started to execute in a resource will complete its execution without any disruption from user activity (such as a keyboard stroke or mouse move) if the desktop machines are used for other purposes. This problem becomes more challenging in a Peer-to-Peer (P2P) model for a desktop grid where there is no central server that decides to allocate a job to a particular resource. This paper describes a P2P desktop grid framework that utilizes resource availability prediction, using group availability data. We improve the functionality of the system by submitting the jobs on machines that have a higher probability of being available at a given time. We benchmark our framework and provide an analysis of our results.

1. Introduction

Volunteer computing is a type of distributed computing in which computer owners donate computing resources (e.g., CPU cycles and storage) for one or more Several initiatives [1–5] in volunteer computing have shown the potential computing power achieved in exploiting the CPU cycles of thousands of machines that are connected to the internet.

Volunteer computing systems often use a master–worker style of computing where tasks are distributed from a master machine (*server*) to worker machines (*volunteers*). The tasks are stored in servers and are sent to the clients for execution. The task is referred to as a *job*. Worker machines register with the server preferences indicating when work can be done on the machine. This information is periodically updated. The server (or in some systems a cluster of servers) maintains all information about projects and volunteers. As systems grow larger this becomes a bottleneck. That has led to investigating the use of peer-to-peer (P2P) architectures for volunteer computing systems (e.g., [6,7]).

Studies (e.g., [8]) suggest that since volunteer computing systems are often less available than controlled clusters that the latter are preferable despite the former being more cost-effective. There are several contributing factors that make it difficult to guarantee the availability of computing resources: (i) resources are donated but not necessarily dedicated, meaning that resources that are available for use change dynamically over time; (ii) high arrival/departure rates mean that it is difficult to provide guarantees about resource availability; (iii) since the donated resources are not dedicated it is possible for the resource's owner to initiate activity that may disrupt the job using the resource.

A P2P environment provides an additional challenge since no resource specific data is stored in a centralized server which makes it difficult to store monitored data needed for resource availability prediction. The primary contribution of this work is to provide a P2P framework for resource selection that incorporates resource availability prediction. In investigating the incorporation of resource availability, we realized that there are environments where resources are not always used by one particular person (e.g., a university computer laboratory). In this case we need to take into account the usage pattern of a group of resources.

This paper is organized as follows. Section 2 presents an architecture and the metrics used to predict future availability. Section 3 describes the implementation details. Section 4 outlines the validation methods. Section 5 describes related work and Section 6 concludes with our observations and future work.

2. Architecture

This section presents the overall architecture of our system (Fig. 1). It has these components: Servers, Client software, Work Units and a Prediction Engine. Volunteers, i.e. desktop machines, are referred to as *peers*.

2.1. Servers

Although resource discovery is P2P, there are several types of servers needed. The *job server* maintains a repository of the jobs

* Corresponding author.
E-mail address: kramach@csd.uwo.ca (K. Ramachandran).

0167-739X/$ – see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.future.2010.10.006

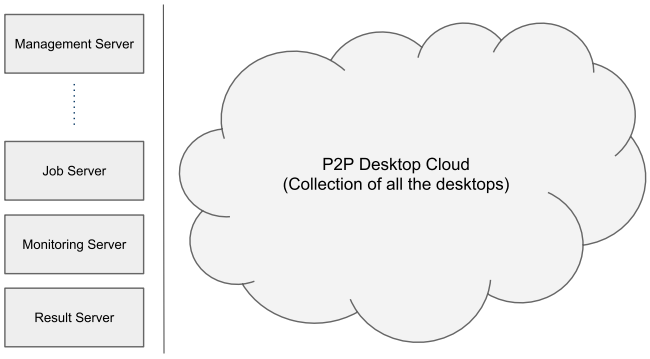

Fig. 1. Architecture of the system.

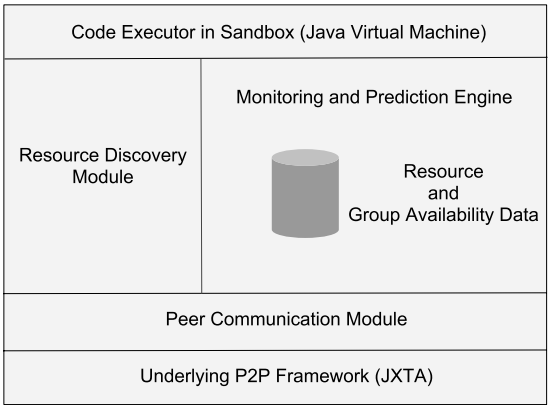

Fig. 2. Client-software components.

to be executed. A client that needs resources to run its tasks makes the request through a job server. A job server executes the resource selection algorithm to find one or more peers and submits the job to selected peers. Information about each job is stored in the job server. A job server can be associated with an organization or with a division of the organization or some other logical units. Thus there can be multiple job servers. The job servers do not maintain information about resources.

A *monitoring server* is used to monitor the jobs submitted by the job servers. There may be multiple monitoring servers associated with a job server. During a job's execution, the *mobile agent* associated with that job sends the status of execution (percentage that is completed) to the monitoring server at a frequency that is predefined when the job is assigned to a resource. The results of a job are sent to a *result server*. The inputs for the mobile agent associated with the job includes the monitoring server and the result server addresses.

A single central management server can be used to authenticate the job servers in the system, so as to avoid non-authorized servers sending jobs to the resources.

2.2. Client software

Each peer has software installed on it. Instances of the software form a self-organizing network in which the resource discovery is fully distributed across the network.

The client software (Fig. 2) consists of the following components:

1. *Communication layer*. The communication layer provides the API for the other layers (Resource Discovery, Code Executor, Monitoring Engine) to send/receive messages with other instances of the client software (peers) and the servers. It is also responsible for the *bootstrapping* of a client. Bootstrapping is the mechanism through which a peer joins the P2P network [9]. This involves finding a member in a network, before joining the network.
2. *Resource discovery*. Resource discovery is done by querying the P2P network for a resource (machine) with particular <attribute_name, attribute_value> pairs associated with it. The query is broadcasted within the P2P network. Software on the matching resources respond to the peer from which the query originated. From the replies received, a specified number of resources are selected. This number is indicated by system parameter: `resourceLimit`. Each resource is sent a `AreYouIdleFor 'n' minutes` message. The list of selected peers is input to the resource selection algorithm which is executed to select the appropriate peer for job execution.
3. *Monitoring engine*. The monitoring engine module records system parameters such as memory usage, CPU usage percentage and user activity to a datastore (lightweight flat-file database) within the peer, at a predefined frequency.

   The Monitoring Engine is also used when the node is executing a job. When the machine is interrupted by user activity, based on the migration policy of the job, it triggers an event in the Code Executor (explained below), which starts the migration process.
4. *Prediction engine*. The data from the local datastore, populated by the monitoring engine, are used as the dataset for learning about the system's behavior.
5. *Code executor*. The code executor module is responsible for the execution of the job in the local machine. The code executor provides a sandbox for the code to execute and protects the system from malicious code. The code executor is implemented by executing the job in a virtual machine [10].

2.3. Work unit

A mobile agent [11] is a software module which moves from node to node autonomously and is executed at each node to which it moves. A job is associated with a mobile agent. This is referred to as a *work unit*. The mobile agent is responsible for migration, communication with a monitoring server and reporting the results to a result server. When a machine is interrupted by user activity while executing the job, instead of terminating the execution, the state is to be transferred to a new machine at which the execution should continue.

2.4. Prediction engine—dedicated and non-dedicated desktops

One of the goals of the volunteer computing architecture is to optimize the usage of desktop resources. These resources can be either individually owned (desktop machines in an office) or used by a group of people (university laboratory machines). In the case of machines that are individually owned, each machine's usage pattern serves as the dataset to predict its availability in the future. However, in a setup like a university laboratory, where the machines are not assigned to a single person, the total laboratory's usage pattern could also be an important criteria for prediction along with the individual machine's usage pattern.

The prediction engine is designed to consider group behavior for predicting availability. An organization can consist of multiple groups (e.g., the university can have multiple laboratories). Each group can have a pattern of resource availability, as can each resource in a group.

The P2P system is used to classify the network into groups $(g_1, g_2, \ldots, g_n)$ where each group represents a collection of resources with a common usage pattern (Fig. 3). When a client requests a resource it searches for a job server. A resource request query is sent from the job server which we will denote by R_{master}.