Heterogeneous Request Process in Large Scale Cloud Environment

A.RAMACHANDRAN,

Assistant Professor, Dept of IT,
Srinivasan Engineering College, Perambalur
msg2chandran@gmail.com

V.VIMALA DHEEKSH

PG Student M.E(CSE)
Srinivasan Engineering College, Perambalur
vimaladheekshanya@gmail.com

Abstract— **The components of the middleware layer run on every processing node of the cloud environment in a decentralized design. To achieve scalability, it envisions that all key tasks of the middleware layer, including estimating global states, placing site modules and computing policies for request forwarding are based on distributed algorithms. Further, it relies on a global directory for routing requests from users on the Internet to access points to particular sites inside the cloud. A gossip protocol P*, executes in a middleware platform and meets the design goals. It provides an optimal solution for a simplified version of the resource allocation problem and an efficient heuristic for the hard problem. The protocol proposed continuously executes, while it's input and consequently its output dynamically changes. Hence to reduce the demand, a time and cost based slot mechanism have been implemented to convert the application into a business oriented application for cloud providers which will be efficient for cloud providers and consumers to minimize the cost of accessing the cloud applications. It will reduce the waiting time of the consumer for accessing the resource in cloud at traffic less environment with efficient cost.**

Key Terms— Middleware platform, Heuristic solution, Resource allocation, gossip protocol.

I.INTRODUCTION

Cloud computing is a popular trend in current computing which attempts to provide cheap and easy access to make the computational resources. Compared to previous paradigms, cloud computing focuses on treating computational resources as measurable and billable utilities. From the clients point of view, cloud computing provides an abstraction of the underlying hardware architecture. This abstraction saves them the costs of design, setup and maintenance of a data center to host their Application Environments (AE). Whereas for cloud providers, the arrangement yields an opportunity to profit by hosting many AEs. This economy of scale provides benefits to both parties, but leaves the providers in a position where they must have an efficient and cost effective data center. This approach centers around a decentralized design whereby the components of the middleware layer run on every processing node of the cloud environment. To achieve scalability, it is envisioned that all key tasks of the middleware layer, including estimating global states, placing site modules and computing policies for request forwarding are based on distributed algorithms.

The core contribution is a gossip protocol P*, which executes in a middleware platform and meets the design goals outlined above. The protocol has two innovative characteristics. First, while gossip protocols for load balancing in distributed systems have been studied before, has no results are available for cases that consider memory constraints and the cost of reconfiguration, which makes the resource allocation problem hard to solve (memory constraints alone make it NP-hard). An optimal solution is provided for a simplified version of the resource allocation problem and an efficient heuristic for the hard problems. Second, the protocol proposed is continuously executes, while its input and consequently its output dynamically changes. Most gossip protocols that have been proposed to date are used in a different way. They assume static input and produce a single output value.

The benefit of a single, continuous execution vs. a sequence of executions with restarts is that in which global synchronization can be avoided and that the system can continuously adapt to changes in local input. On the other hand, its drawback is that the behavior of a protocol with dynamic input is more difficult to analyze. Also, the cost of the system to react to a high rate of change in local output can potentially be higher than implementing a set of changes after each synchronized run. Based on the work thus far, it is believed that, for a gossip protocol running in large-scale dynamic environments, the advantages of continuous execution with dynamic input outweigh its potential drawbacks.

II.SYSTEM MODEL

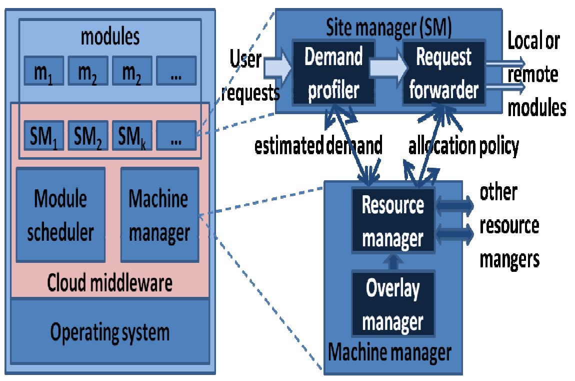


Fig 1.SYSTEM DESIGN

A cloud environment spans several datacenters interconnected by an internet. Each of these datacenters contains a large number of machines that are connected by a high-speed network. Users access sites hosted by the cloud environment through the public Internet. A site is typically accessed through a URL that is translated to a network address through a global directory service, such as DNS. A request to a site is routed through the Internet to a machine inside a datacenter that either processes the request or forwards it. In this paper, we restrict ourselves to a cloud that spans a datacenter containing a single cluster of machines and leave for further work the extension of our contribution to an environment including multiple datacenters.

Each site manager handles user requests to a particular site. It has two important components: a demand profiler and request forwarder. The demand profiler estimates the resource demand of each module of the site based on the request statistics, QoS targets, etc This estimate is forwarded to all machine managers that run instances of modules belonging to this site. Similarly, the request forwarder sends user requests for processing to instances of modules belonging to this site. Request forwarding decisions take into account the resource allocation policy and constraints such as session affinity. Figure shows the components of a site manager and how they relate to machine managers. The remainder of this paper focuses on the functionality of the resource manager component.

III. FORMALIZING THE PROBLEM OF RESOURCE ALLOCATION BY THE CLOUD MIDDLEWARE

The specific problem addressed is that of placing modules (more precisely: identical instances of modules) on machines and allocating cloud resources to these modules, such that a cloud utility is maximized under constraints. As cloud utility we choose the minimum utility generated by any site, which we define as the minimum utility of its module instances. We formulate the resource allocation problem as that of maximizing the cloud utility under CPU and memory constraints. The solution to this problem is a configuration matrix that controls the module scheduler and request forwarder components. At discrete points in time, events occur, such as load changes, addition and removal of site or machines, etc. In response to such an event, the opti- mization problem is solved again, in order to keep the cloud utility maximized. We add a secondary objective to the optimization problem, which states that the cost of change from the current configuration to the new configuration must be minimized.

A.THE MODEL

We model the cloud as a system with a set of sitesS and a set of machines N that run the sites. Each site s $\in$ S is composed of a set of modules denoted by M_s .We consider a system that may run more than one instance of a module m, each on a different machine, in which case its CPU demand is divided among its instances. The demand $\omega_{n,m}(t)$ of an instance of m run- ning on machine n is given by $\omega_{n,m}(t) = \alpha_{n,m}(t)\omega_m(t)$, where

$n \in N$ $\alpha_{n,m}(t) = 1$ and $\alpha_{n,m}(t) \geq 0$.

It is called that the matrix A with elements $\alpha_{n,m}(t)$ the configuration (matrix) of the system. A is a non-negative matrix with $1^T A = 1^T$

A machine n $\in$ N in the cloud has a CPU capacity Ω_n and a memory capacity Γ_n . We use Ω and Γ to denote the vectors of CPU and memory capacities of allthe machines in the system. An instance of module m running on machine n demands $\omega_{n,m}(t)$ CPU resource and γ_m memory resource from n. Machine n allocates to module m the CPU capacity $\hat{\omega}_{n,m}(t)$ (which may be different from $\omega_{n,m}(t)$) and the memory capacity γ_m . We define the utility $u_{n,m}(t)$ generated by an instance of module m on

A.Ramachandran, V.Vimala Dheeksh

machine n as the ratio of the allocated CPU capacity to the demand of the instance on that particular machine. Wefurther define the utility of a module m as

$$u_m(t) = \min_{n \in N} \{u_{n,m}(t)\}$$

and that of a site as the minimum of utility of its modules. Finally, the utility of the cloud U^c is the minimum of the utilities of the sites it hosts. As a consequence, the utility of the cloud becomes the minimum utility of any module instance in the system.

B.THE OPTIMIZATION PROBLEM

For the above model, we consider a cloud with CPU capacity Ω, memory capacity Γ, and demand vectors ω, γ . We first discuss a simplified version of the problem. It consists of finding a configuration A that maximizes the cloud utility $U^{c.}$

Maximize $U^c(A, \square)$ OP(1)

subject to $A \geq 0, \quad 1^T A = 1^T$ (a)

$\hat{\Omega}(A, \omega)1 \quad \Omega$ (b)

Our concept of utility is max-min fairness and our goal is to achieve fairness among sites. This means that we want to maximize the minimum utility of all sites, which we achieve by maximizing the minimum utility of all module instances.

Constraint (a) of OP(1) relates to dividing into shares the CPU demand of each module into the demand of its instances. The constraint expresses that all shares are non-negative and add up to 1 for each module.

maximize $U^c(A(t+1), \omega(t+1))$

minimize $c^{\square}(A(t), A(t+1))$

subject to

$$A(t+1) \geq 0, \ 1^T A(t+1) = 1^T$$
$$\hat{\Omega}(A(t+1), \omega(t+1))1 \quad \Omega$$
$$\mathrm{sign}(A(t+1))\gamma \quad \Gamma.$$

(OP(2))

This optimization problem has prioritized objectives in the sense that, among all configurations A that maximize the cloud utility, we select one that minimizes the cost function c. While this paper considers only events in form of changes in demand, OP(2) allows us to express (and solve) the problem of finding a new allocation after other events, including adding or removing sites or machines.

IV. THE PROTOCOL FOR DISTRIBUTIVE RESOURCE ALLOCATION

In this section, we present a protocol P, which is a heuristic algorithm for solving OP(2) and which represents our proposed protocol for resource allocation in a cloud environment. P is a gossip protocol and has the structure of a round-based distributed algorithm (whereby round-based does not imply that the protocol is synchronous). When exe- cutting a round-based gossip protocol, each node selects a subset of other nodes to interact with, whereby the selection function is often probabilistic. Nodes interact via 'small' messages, which are processed and trigger local state changes. In this work, node interaction follows the so-called push-pull paradigm, whereby two nodes exchange state information, process this information and update their local states during a round.

P runs on all machines of the cloud. It is invoked at discrete points in time, in response to a load change. The output of the protocol, the configuration matrix A, is distributed across the machines of the system. A controls the start and stop of module instances and determines the control policies for module schedulers and request forwarders. The protocol executes in the resource manager components of the middleware architecture. A set of candidate machines to interact with is maintained by the overlay manager component of the machine manager. We assume that the time it takes for P to compute a new configuration A is small compared to the time between events that trigger consecutive runs of the protocols. At the time of initialization, P reads as input a feasible configuration of the system (see below). At later invocations, the protocol reads as input the configuration matrix produced during the previous run.

A. Functionalities the protocol P Uses

a) Random selection of machines: P relies on the ability of a machine to select another machine of the cloud uniformly at random. In this work, we approximate this ability by using CYCLON, an overlay protocol that produces a time-varying network graph with properties of a random network [3].

b) Resource allocation and module scheduling policy.

c) Computing a feasible configuration: P requires a feasible configuration as input during its initialization phase. A simple greedy algorithm can be used for this purpose, which we present in [4] due to space limitation.

B. Protocol P': An Optimal Solution to OP(1)

We developed the protocol P', which is a distributed solution to OP(1). P' is a gossip protocol that produces a sequence of configuration matrices

$A(r), r = 1, 2, \ldots,$

such that the series of cloud utilities $U^c(A(r), \omega)$ con- verges exponentially fast to the optimal utility. Due to space limitation, P' is described and its properties proved in [4]. We would encourage the reader to look up this protocol, as it is quite simple and enables a better understanding of P, which can be seen as an extension of P'. During each round of P', two machines perform an equalization step whereby CPU demand is moved from one machine to another machine in such a way that their relative demands are equalized.

C. Protocol P: A Heuristic Solution to OP(2)

OP(2) differs from OP(1) in that memory constraints of individual machines are considered and a secondary objective is added for the purpose of minimizing the cost of adapting the system from the current to a new configuration that maximizes the utility for the new demand. Introducing local memory constraints to the optimization problem turns OP(1), which we showed can be efficiently solved for many practical cases [4], into an NP-hard problem [2].

P employs the same basic mechanism as P' as it attempts to equalize the relative demands of pairs of machines during a protocol round. Due to the local memory constraints, such a step does not always succeed.

P uses the following approach to achieve its objectives. First, pairs of machines that execute an equalization step are often chosen in such a way that they run instances of common modules. To support this concept, we maintain on each machine n the set N_n of machines in the cloud that run module instances common with n. To avoid the

possibility of the cloud being partitioned into disjoint sets of interacting machines, n is occasionally paired with a machine outside of the set N_n to execute an equalization step. This dual approach keeps low the need for starting new module instances and thus keeps the cost low. Second, during an equalization step, P attempts to reduce the difference in relative demand between two machines, in case it cannot equalize the demand. Further, P attempts to execute an equalization step in such a way that the demand for a specific module is shifted to one machine only. This concept aims at increasing the probability that an equalization step succeeds in equalizing the relative demands, thus increasing the cloud utility.

The pseudo code of P is given in Algorithm 1. To keep the presentation simple, we omit thread synchronization primitives which prevent concurrent machine to machine interactions. Note that setting $\alpha_{n,m} = 0$ implies stopping module m on machine n.

During the initialization of machine n, the algorithm reads the CPU demand vector, the CPU and memory capacity vectors, and the row of the configuration matrix for n. (For an efficient implementation, n must only read those vector components that refer to itself and its module instances.) Then, it starts two threads: an active thread, in which the machine periodically executes a round, and a passive thread that waits for another machine to start an interaction.

Algorithm 1 Protocol P computes a heuristic solution for OP(2) and returns a configuration matrix A

```
Initialization
 1: read $\omega, \Omega, \Gamma, row_n(A), N_n$;
 2: start the passive and active threads
active thread
 3: for $r = 1$ to $r_{max}$ do
 4:   if rand(0..1) < p then
 5:     choose $n^0$ at random from $N_n$;
 6:   else
 7:     choose $n^0$ at random from $N - N_n$;
 8:   send($n^0, row_n(A)$);
        $row_{n^0}(A)$ =receive($n^0$);
 9:   equalizeWith($n^0, row_{n^0}(A)$);
10:   sleep(roundDuration);
11: write $row_n(A)$;
passive thread
12: while true do
13:   $row_{n^0}(A)$=receive($n^0$);
      send($n^0, row_n(A)$);
14:   equalizeWith($n^0, row_{n^0}(A)$);

proc equalizeWith(j, $row_j(A)$)
 1: $l = \arg\max\{v_n, v_j\}$; $l^0 = \arg\min\{v_n, v_j\}$;
 2: if $j \in N_n$ then
 3:   moveDemand1($l, row_l(A), l^0, row_{l^0}(A)$);
 4: else
 5:   moveDemand2($l, row_l(A), l^0, row_{l^0}(A)$);
```

```
proc  moveDemand1(l, row_l (A), $l^0$, row_{l^0} (A))
 1:  compute $\Delta\omega$ such that
     $\frac{1}{\Omega_l}(\sum_m \omega_{l,m} - \Delta\omega) = \frac{1}{\Omega_{l^0}}(\sum_m \omega_{l^0,m} + \Delta\omega)$
 2:  let mod be an array of all modules
     that run on both l and $l^0$,
     sorted by increasing $\omega_{l,m}$
 3:  for i = 1 to |mod| do
 4:    m = mod[i]; $\delta\omega = \min(\Delta\omega, \omega_{l,m})$;
 5:    $\Delta\omega$ −= $\delta\omega$; $\delta\alpha = \alpha_{l,m} \frac{\delta\omega}{\omega_{l,m}}$;
       $\alpha_{l^0,m}$ += $\delta\alpha$; $\alpha_{l,m}$ −= $\delta\alpha$;
```

```
proc  moveDemand2(l, row_l (A), $l^0$, row_{l^0} (A))
 1:  compute $\Delta\omega$ such that
     $\frac{1}{\Omega_l}(\sum_m \omega_{l,m} - \Delta\omega) = \frac{1}{\Omega_{l^0}}(\sum_m \omega_{l^0,m} + \Delta\omega)$
 2:  let mod be an array of all modules
     that run on l, sorted by decreasing
     $\frac{\omega_{l,m}}{\gamma_m}$;
 3:  for i = 1 to |mod| do
 4:    m = mod[i]; $\delta\omega = \min(\Delta\omega, \omega_{l,m})$;
 5:    if $\gamma_m + \sum_{i|\alpha_{l^0,i}>0} \gamma_i \le \Gamma_{l^0}$ then
 6:      $\Delta\omega$ −= $\delta\omega$; $\delta\alpha = \alpha_{l,m} \frac{\delta\omega}{\omega_{l,m}}$;
         $\alpha_{l^0,m}$ += $\delta\alpha$; $\alpha_{l,m}$ −= $\delta\alpha$;
```

The active thread executes r_{max} rounds. In each round, n chooses a machine n^0 uniformly at random from the set N_n with probability p and from the set $N - N_n$ with probability $1 - p$. Then n sends its state (i.e., row_n (A)) to n^0, receives n^0's state as a response, and calls the procedure equalizeWith(), which performs the equalization step. The passive thread executes in a continuous loop. Whenever n receives the state from another machine n^0, it responds by sending its own state to n^0 and performing an equalization step by invoking equalizeWith().

The procedure equalizeWith() attempts to equal- ize the relative demands of machines n and n^0. It first identifies the machine l with the larger (or equal) relative demand and the machine l^0 with the lower relative demand. Then, if n^0 belongs to N_n and thus runs at least one common module instance, procedure moveDemand1() is invoked. Otherwise moveDemand2() is invoked. moveDemand1() equalizes (or reduces the differ- ence) of the relative demands of the two machines, by shifting demand from the machine l with the larger relative demand to the machine l^0 with the smaller relative demand. It starts by computing the demand $\Delta\omega$ that needs to be shifted from l to l^0 (step 1). Then, from the set of modules that run on both machines, taking an instance with the smallest demand on l, it proceeds to shift the demand from l to l^0, until a total of $\Delta\omega$ demand is shifted, or it has exhausted the set of modules. moveDemand2() equalizes (or reduces the differ- ence) of the relative demands of the two machines, by moving demand from the machine with larger rel- ative demand to the machine with smaller relative de- mand. Unlike moveDemand1(), moveDemand2() starts one or more module instances at the destination machine, to move demand from the source machine to the destination, if sufficient memory at the destination machine is available. Finding a set of instances at the source that equalize the relative demands of the partic- ipating machines while observing the available memory of the destination is a Knapsack problem. A method called greedy approximation is applied, whereby the module m with the largest value of $\frac{\omega_{l,m}}{\gamma_m}$ is moved first, followed by the second largest, etc., until the relative demands are equalized or the set of candidate modules is exhausted.

V. PRICE AND TIME-SLOT NEGOT IATIONS

The *PTN* mechanism consists of the following: 1) an aggregated utility function; 2) negotiation strategies; and 3) a negotiation protocol.

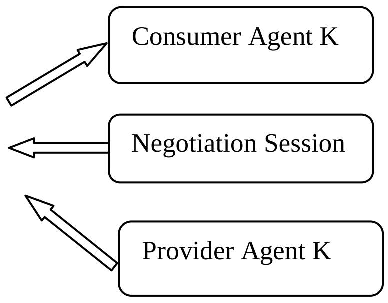
A. Utility Functions

A utility function *U*(*x*) represents an agent's level of satisfaction for a negotiation outcome *x*. Since each Cloud participant has different preferences for different prices and time slots, a price utility function, a time-slot utility function, and an aggregated utility function are used to model the preference ordering of each proposal and each negotiation outcome.

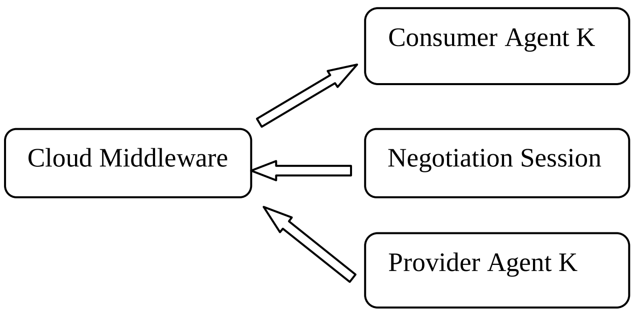
B. Negotiation Strategy

This work considers bilateral negotiations between a consumer and a provider, where both agents are sensitive to time and adopt a time-dependent concession-making strategy for *PTNs*. Since both agents negotiate on both price and time slot, generating a counterproposal can be making either a concession or a tradeoff between price and time slot. Hence, an agent's strategy for multi-issue negotiation is implemented using both the following: 1) a tradeoff algorithm and 2) a concession making algorithm.

C. Negotiation Protocol



The negotiation protocol of the PTN mechanism follows, Rubinstein's alternating offers protocol in which agents make counteroffers to their opponents in alternate rounds. Both agents generate counteroffers and evaluate their opponent's offers until either an agreement is made or one of the agents' deadline is reached. If a counterproposal is accepted, both agents found a mutually acceptable price and time slot. If one of the agents' deadline expires before the yreach an agreement, the negotiation fails.



VI.RELATEDWORK

The problem of application placement in the context of resource management for datacenters has been studied before (e.g., [2], [7]), and solutions are already available in middleware products [8]. While these product solu- tions allow for a fair resource allocation in a similar way as our scheme does, they rely on centralized archi- tectures, which do not at all scale to system sizes we consider in this paper.

Distributed load balancing algorithm have been extensively studied for homogeneous as well as hetero- geneous systems, for both divisible and indivisible demands. These algorithms typically fall into two classes: diffusion algorithms (e.g., [11]) and dimension exchange algorithms (e.g., [12]). Convergence results for different network topologies and different norms (that measure the distance between the system state and the optimal state) have been reported, and it seems to us that the problem is well understood today. The key difference to the problem addressed in this paper is that these algorithms do not take into account memory constraints. Considering memory constraints makes the problem NP- hard and does require a new approach.

VII CONCLUSION

With this paper, we make a significant contribution towards engineering a resource management middleware for a site-hosting cloud environment. We identify a key component of such a middleware and present a protocol that can be used to meet our design goals for resource management: fairness of resource allocation with respect to sites, efficient adaptation to load changes and scalability of the middleware layer in terms of both the number of machines in the cloud as well as the number of hosted sites.

We presented a gossip protocol, that Computes the heuristic solution to the resource allocation problem and evaluated its performance. In all the scenarios we investigated, we observe that the protocol qualitatively behaves as expected based on its design. Regarding fairness, the gossip protocol performs close to an ideal system for scenarios where the ratio of the total memory capacity to the total memory demand is large. The simulations suggest that the protocol is scalable in the sense that all inves- tigated metrics do not change when the system size (i.e., the number of machines) increases proportional to the external load (i.e., the number of sites). Note that if we would solve the resource allocation problem expressed in OP(2) by P in a centralized system, then the CPU and memory demand for that resource allocation system would increase linearly with the system size. Another novelty of this work is formulating a novel time-slot utility function that characterizes preferences for different time slots. These ideas are implemented in an agent based Cloud testbed. This strongly suggests to us that a centralized solution for the problem we address in this paper will not be feasible.

The results reported in this paper are Building blocks towards engineering a resource management solution for large-scale clouds. Pursuing this goal, we plan to address the following issues in future work: (1) Develop a distributed mechanism that efficiently places new sites. (2) Extend the middleware design to become robust to machine failures. (3) Extend the middleware design to span several clusters and several datacenters, while keeping module instances of the same site "close to each other", in order to minimize response times and communication overhead.

REFRENCES

[1] Adam and R. Stadler, '*Service middleware for self-managing large scale systems,*' IEEE Trans. Network and Service Management, vol. 4, no. 3, pp. 50–64, Apr. 2008.

[2] C. Tang, M. Steinder, M. Spreitzer, and G. Pacifici, '*A scalable application placement controller for enterprise data centers,*' in 2007. International Conference on World Wide Web.

[3]D. Carrera, M. Steinder, I. Whalley, J. Torres, and E. Ayguade, '*Utility based placement of dynamic web applications with fairness goals,*" in 2008 IEEE Network Operations and Management Symposium.

[4]E. Loureiro, P. Nixon, and S. Dobson, '*Decentralized utility maximization for adaptive management of shared resource pools,*' in 2009 International Conference on Intelligent Networking and Collaborative Systems.

[5]F. Wuhib, M. Dam, R. Stadler, and A. Clem, '*Robust monitoring of network-wide aggregates through gossiping,*' IEEE Trans. Network and Service Management, vol. 6, no. 2, pp. 95–109, June 2009.

[6]F. Wuhib, M. Dam, and R. Stadler, '*A gossiping protocol for detecting global threshold crossings,*' IEEE Trans. Network and Service Management, vol. 7, no. 1, pp. 42–57, Mar. 2010.

[7]Fetahi Wuhib, Rolf Stadler, and Mike Spreitzer, '*A Gossip Protocol for Dynamic Resource Management in Large Cloud Environments,*' IEEE transactions on network and service management, vol. 9, no. 2, June 2012

[8]M. Jelasity, A. Montresor, and O. Babaoglu, '*Gossip-based aggregation in large dynamic networks,*' ACM Trans. Computer Syst., vol. 23, no. 3, pp. 219–252, 2005.

[9] Mark Jelasity, Ozalp Babaoglu, '*T-Man: Fast Gossip-based Construction of Large Scale Overlay Topologies,*' Technical Report UBLCS-2004-7

[10] R. L. Graham, '*Bounds on multiprocessing timing anomalies,*' SIAM J. Applied Mathematics, vol. 17, no. 2, pp. pp. 416–429, 1969.

[11] S. Voulgaris, D. Gavidia, and M. van Steen, '*CYCLON: inexpensivemembership management for unstructured p2p overlays,*' J. Network and Systems Management, vol. 13, no. 2, pp. 197–217, 2005.