A DECENTRALIZED LOCATION SERVICE

Applying P2P technology for picking replicas on replicated services

Luis Bernardo, Paulo Pinto
Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, P-2829-516 Caparica, Portugal

Keywords: Scalable Internet Services and Applications, location service, self-adaptable service, peer-to-peer

Abstract: Scalable Internet services are based on sets of peer application servers. A decentralized location service is used to resolve human readable application identifiers and return the nearest application server reference. This paper evaluates several services and algorithms from the Internet, grid and peer-to-peer community services. It identifies two potential problems and proposes a new approach for handling them. Existing techniques structure the overlay networks using tree structures. The proposed service enhances the structure with meshed structures at each level, creating dynamically multiple paths to enhance scalability. We present a study and simulation results on one aspect of scalability – sudden load of requests from users. Our service adapts to the load reaching a stable stage and performing resolution requests before a certain time limit.

1 INTRODUCTION

The growing number of users, computers, and applications servers is rapidly driving Internet to a new reality. An Internet application can no longer be a single server running on a single node. Applications must be supported by peers of machines. The convergence of web services and grid technology (Foster, 2002) provides a hint for what might be the future Internet applications.

An open architecture for applications must necessarily detach the identification of an application from the location of the application servers, in order to cope with a huge community of users and a large dynamic set of servers (peers). Such applications must not be identified by an IP address has present URLs are. Instead, an intermediate identifier must be used to bridge a human significant representation to the actual servers' location. This introduces the necessity for a middleware service that resolves the intermediate identifier to an application server reference.

This paper focuses on the implementation of such a location service. The location service must adapt to the dynamics of the application server peers and provide a scalable service to the applications. Section 2 presents an overview of the envisioned scenario. On section 3 we review several proposals of location-like services from different communities. We analyze their ability to handle simultaneous peaks of lookups and updates, and the resulting load distribution on the network. On section 4 and 5, we present our location service proposal. The paper describes its architecture and algorithms, in light of the same requirements. We show that the introduction of a dynamic structure provides a significant improvement over other approaches. The location service performance is evaluated using a set of simulations under loaded conditions on section 6.

2 PROBLEM DEFINITION

This paper assumes the existence of an active network, with a ubiquitous set of compute nodes, where application and middleware servers may run. A grid middleware layer enables the dynamic deployment of application and service servers on demand, and the access to network resources, including bandwidth and processing power. Notice though, that Internet is not homogeneous. We assume that it is composed by several interconnected high-bandwidth core networks, which interconnect a huge number of high-bandwidth and lower bandwidth networks. In order to avoid bottlenecks at the core, communication should be localized.

The location service is one of the key components for the future Internet scalability. The location service must provide an anycast resolution (Partridge, 1993) for the intermediate application identifiers. It must return the location of the application server nearest to the client. The actual

Bernardo L. and Pinto P. (2004).
A DECENTRALIZED LOCATION SERVICE - Applying P2P technology for picking replicas on replicated services.
In *Proceedings of the First International Conference on E-Business and Telecommunication Networks*, pages 39-47
DOI: 10.5220/0001384700390047
Copyright © SciTePress

metrics changes depending on the location service and on the application requirements, but it may include hops, bandwidth, stability of the nodes, processing power, etc.

The location service creates an overlay network on top of the compute nodes, which supports the application server lookup operation. The user preferred applications evolve in time. For instance, a local news service may become a top news application due to a notorious event (e.g. a local elections tie or an accident), or an e-commerce site may jump to the top due to aggressive marketing. A huge jump on the preference order may produce a huge increment on the number of clients (n_2/n_1 if Zipf distribution (Adamic, 2002) is followed). This will lead necessarily to the increase on the number of servers to cope with the demand (e.g. Content Delivery Network applications (Vakali, 2003) distribute replicas of pages to handle load peaks). A generic algorithm was proposed in (Bernardo, 1998) to control the replica deployment. The location service must be able to handle this peak of updates, and in parallel, the concurrent peak of lookups. Centralized approaches, based on a home location server may fail due to a peak of millions of requests. Caching solutions may also fail, because they may conceal the appearing of new application server replicas.

The envisioned location service provides two operations: *lookup*(*id, range*) and *update*(*id*, *serv_reference, range*). Each application server registers on the location service its reference associated with a unique application identifier (*id*) for a certain range. Clients search for one or more replicas within a range on the network.

3 LOCATION-LIKE SERVICES

Several existing services support the location service required functionalities. They differ on how lookups are performed: either use flooding (broadcast when available) or guided search.

Flooding approaches are common for micro-location services (e.g. Jini (Gupta, 2002)), for unstructured peer-to-peer (P2P) networks (e.g. Gnutella), and for routing algorithms in Ad Hoc networks (e.g. AODV (Perkins, 2003)). Updates are made on a local node, resulting on random information distribution. A flooding approach does not require (almost) any setting up, and adapts particularly well to unstable networks, unstable data and unstable nodes. However, it has high search costs and does not scale with the increase of the number of clients and of the lookup range (Schollmeier, 2002). Therefore, it is not adapted to provide a global view of a system. Strategies for reducing the lookup costs include (Chawathe, 2003): the creation of supernodes; the replication of information on neighbor nodes; the use of selective flooding to reduce the number of messages; and the control of the message flow. Supernodes create centralization points on a distributed network, which inter-connect lower power and more unstable nodes. They define a backbone that carries most of the flooded messages. In result, a small world effect is created that reduces the range needed to run lookups. However, supernodes also create concentration points, which can become a bottleneck on the system through link and server saturation or the increased message delay in result of flow control. Replication of *id* information distributes the load through several nodes. When replication is done at supernodes (e.g. a Clip2 Reflector replicates information for all subordinate nodes), it restricts flooding to a second hierarchical layer (connecting supernodes) with a slight increase in update costs (see table 1).

On the other hand, guided search approaches create an *id* table. Updates and lookups are made on nodes dedicated to that *id*, selected using operation *route(id)*. The table can be kept on a centralized node or partitioned and distributed on several nodes. Centralized approaches (e.g. Napster) simplify routing but introduce a single point of failure that can slow down the entire system. The performance of distributed approaches depends on the structure of *id* and on the geometry of the overlay network defined by the nodes (Gummadi, 2003). The distributed approaches include the big majority of naming and routing services and structured P2P.

DNS is a good example of the first group. DNS relies on a hierarchical structure of nodes matched with the identifier hierarchy. This approach simplifies routing because the name completely defines the resolution path. If h is the maximum hierarchical level, it has a maximum length of $2h-1$. However, it contains most of the centralized approaches limitations, benefiting only from the information fragmentation over several nodes. DNS improves its scalability using extensively caching and node replication. Caching reduces the amount of information exchanged amongst peers but prevents the use of DNS when referring to moveable or on-off entities. It was not a requirement at the time because IP addresses did not change frequently. The inflexibility of DNS routing (a single path towards the node with the required *id*) dwarfs the effects of node replication. The localization of *id* resolution (the selection of the nearest replica) is only supported by DNS extensions (e.g. Internet2 Distributed Storage Information (Beck, 1998)).

Structured P2P are based on distributed hash tables (DHT). Location servers (nodes) and

Table 1: Summary of services features: (*neighbours*) number of neighbours, (*search*) search costs, (*#path*) maximum number of independent paths available, (*update*) updates costs and (*Join*) node insertion costs. n and n_s are the average number of neighbours. N is the toal number of nodes. b and L are algorithm parameters.

	neighbours	search	# paths	update	Join
Gnutella	n	N	Path(N)	1	n
Gnut. supernode k agregation	n_s for supernodes	N/k	Path(N/k)	2	$n+1$ for nodes n_s+k for supernodes
DNS - h levels	1 node above	$\leq 2h$-1	1	$\leq 2h$-1	1 for leafs
Pastry	$O(b.\log_b N+L)$	$O(\log_b N)$	$\log_b N$	$O(\log_b N)$	$O(b.(\log_b N)^2)$
Tapestry	$O(b.\log_b N)$	$O(\log_b N)$	$\log_b N$	$O(\log_b N)$	$O(b.\log_b N)$
Brogade with k Aggregation	$O(b.\log_b k+1)$ nodes $O(b.\log_b N)$ supern.	$O(\log_b (Nk))$	1 for long range	$O(\log_b N)$	$O(b.\log_b k)$ for nodes $O(b.\log_b k+b.\log_b(N/k))$

registrations are mapped to identifiers (*ids*), often calculated using hash functions. Nodes keep registrations for a subset of the *id* space. They distribute routing information creating self-organizing node structures, which exhibit some hierarchical characteristics. Each node behaves as a classical root for its local *ids*. Structured P2P services that support localization on the resolution of *id* for replicated objects include: Pastry (Rowstron, 2001), Tapestry (Hildrun, 2002), Brogade (Zhao, 2002), and other algorithms derived from Tapestry (e.g. Kademlia, AGILE).

Pastry and Tapestry are based on similar approaches. They both use strings of digits of base b as *Ids* (with a maximum N) and organize the overlay network in multiple trees (one for each *Id*). Pastry structure is a little bit more complex because it adds a complementary ring structure (L pointers), for reliability and for improving the last routing hops. However, the main routing scheme for Pastry is based on a tree. Each node is a root for the local *Ids* tree. The root connects to nodes which differ only in the last *id* digit. Successive layers differ on increasing number of digits. Nodes are members of several *id* trees in different layers. Each node maintains a routing table with $log_b N$ columns (one for each hierarchical level – related to the number of shared digits) and b rows (one for each digit value). Nodes route *ids* following the tree from the starting node to the root node, resulting in a maximum path length of $log_b N$ steps. Tapestry optimizes lookup locality for replicated sets by disseminating pointers to the application servers on the path from the node with the server till the root (associated with the *id*). Lookup is done following the direction to the *id*'s root, until a first registration is found. Pastry only replicates registrations on the direct neighbors of the root node for an *id* tree, producing longer resolution paths and less precise localization.

Brogade proposes the use of two layers of P2P overlay networks, running independent Tapestry services. Nodes with higher bandwidth connections and higher processing power are promoted to supernodes. Supernodes collect information about the lower layer node *ids* inside their region, and treat them as data on their layer. Brogade uses hierarchical routing with two levels. Users benefit from the use of more powerful links on connections. However, due to the pure hierarchy, supernodes create centralized points of failure and preclude the use of other lookup paths on the lower hierarchy level. The localization properties of Tapestry may also degrade due to the two-level routing. Local lookups use only the lower layer, but longer lookups go through the higher Tapestry network layer, resulting on a total path of $O(2log_b(k)+log_b(N/k))$.

Globe grid location service (Steen, 1998) also proposed an overlay tree structure. Globe location service trades off update flexibility for a bigger resolution path. Instead of complete information, nodes store forward pointers to other nodes (hints), which define a chain pointing to the nodes with the information. Hints usage reduces update costs because updates are propagated only to a level where another registration exists. On structured P2P updates are always propagated to the root node. Globus grid information service (Czajkowski, 2001) adopted a different approach: each node provides complete information about resources on a set of compute nodes (named a virtual organization). Nodes are organized hierarchically, feeding data upwards the hierarchy using a data access protocol and an event service. Nevertheless, if the system becomes very dynamic their choice may fail due to the update overhead.

Can the existing services support dynamic application server creation and peaks of user demand? For very dynamic information (e.g. Ad Hoc networks or fast moving objects) and for limited ranges, flooding approaches are capable of fulfilling all requirements (see update and join costs in table 1). For a scalable large scale view guided-search approaches must be adopted but they fail if the information dynamics is not slow enough. During lookup load peaks the maximum number of lookups supported depends on the number of nodes

with the information for the *id*, their capacity, but also on the total number of paths available to route to those nodes. If a concurrent update peak is also running, it also depends on update cost and on how fast updates are available to the users. Most of the structured P2P systems propose the use of information replication on neighbor nodes, and caching of previous lookup results (useless for this problem) to handle overload. However, they do not define an algorithm to dynamically perform the replication, requiring some form of external configuration. They also suffer from an intrinsic limitation of a tree organization: **it concentrates too much load on the root node**. Figure 1 illustrates the problem. If the top hierarchical layer has *k* branches, from which, *n* branches have the (application) *id* registered, then the peak lookup load on the root node is $Q\left(1-n\cdot k^{-1}\right)$ for a uniformly distributed global lookup load of Q queries per second. This means that a constant fraction of the load will be handled by the root node. This paper proposes a solution to these two problems.

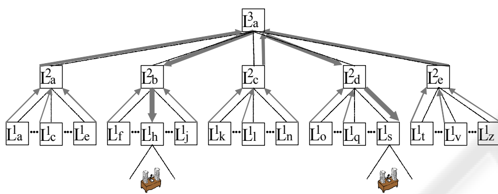

Figure 1: Load distribution on a pure tree structure

4 LOCATION SERVICE ARCHITECTURE

This paper proposes a service location based on a dynamic tree, although enhanced with meshed structures connecting the nodes at each hierarchical level. For low load levels, the location service operates using only the paths defined by the tree. For higher load levels, extra horizontal paths are added, increasing the number of paths until the maximum supported by the overlay network. When the load is really intense new location servers can be created to split the load, and possibly, a new layer of the hierarchy can be created. This paper proposes an algorithm to control the activation of the meshed paths and the replication of nodes, based on load measurements.

The lowest layer forms a static meshed network of simple proxy entities (Local Proxy – LP) responsible to both connect to the location service and maintain neighborhood relationships. It forms a topological grid to provide a sense of "space" to the system (can be physical space, something related with availability of bandwidth between servers, etc.). Local Proxies have complete knowledge of the entities in their fixed region (be it an active node, or a set of nodes). They can resolve the identifier using the upper layers.

The layers above are composed of location servers, named L. L servers usually have hints pointing to another L server, but may have complete registration information on their first hierarchical level. L server overlay network structure is created dynamically based on the maximum range specified on *update* operations and on the load (see below). We assume that an L server can be dynamically dispatched on a particular compute node. The hierarchy is created using the clustering algorithm presented in (Bernardo, 1998b), which runs the highest hierarchical level L servers on the more resourceful compute nodes (inline with the supernode approach). As long as there are global range application servers registered, a hierarchical tree structure exists covering the entire network. Otherwise, if all applications are regional or local, there are only several independent trees. Clients can still locate *ids* using a flooding approach on the various relative roots (limited by the *range* parameter). A meshed structure connects the L servers of a tree at each hierarchical level, except the first L-server hierarchical level, which connects the entire network. But, as table 1 shows the flooding approach does not scale for a large number of clients. Therefore, we assume that application servers always specify the entire range where their clients will come from.

5 ADAPTATION OF THE RESOLUTION PATH

When an application server registers its reference, LP forwards its registration information to the first level L server. This L server disseminates a hint up the tree in direction to the root creating a single vertical path to resolve the *id*. Vertical dissemination stops when a hierarchical layer node (*h*) which embraces the required number of LPs specified in the range or when another replica hint for the same *id* is found.

A. SpreadRange

For handling the root overload problem, L servers may also disseminate hints horizontally, creating extra paths that reduce the lookup load on L servers at higher hierarchical layers. The scope of the horizontal dissemination area is defined by the

parameter SpreadRange. For instance, in fig. 2, if L^2_d SpreadRange for the illustrated application server includes L^2_e, then L^3_a stops answering to lookups coming from L^2_e (see fig. 1).

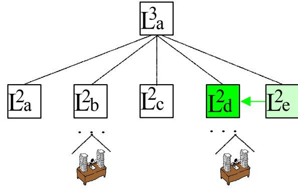

Figure 2: SpreadRange parameter

L servers experiencing overload control horizontal dissemination for the immediately lower layer L servers, using SpreadRange Control Messages: they may send a request to some of their lower layer L servers to increment or reduce the SpreadRange on a set of identifiers. Each receiver tests its maximum range and local load, and may refuse an increase if they are higher than the maximum values allowed. When an identifier is first registered, no horizontal dissemination is used, unless a hint of an existing replica has been received from a neighbor L server. The SpreadRange parameter at an L server will be increased or decreased in result of increases and decreases of the lookup load coming from near L servers. The rationale is to deploy the structure with the lowest update overhead, yet adapted to the lookup load.

If several application server replicas are available on neighbor regions, L servers should reply to the lookups distributing the load amongst the application servers taking into account the "distance" to each of the replicas. On this case, all L servers must use the same SpreadRange value, to guarantee a balanced load distribution.

B. CoreRange

Horizontal hint dissemination can still not solve the problem because the load is still concentrated on the branch linking to the LP of the application server. Therefore, a stronger form of horizontal dissemination was introduced: the cloning of the exact L server information (and not hints) about a set of "hot" *ids* on the neighbor L servers, called replicates. The scope of replicate horizontal dissemination is controlled by the parameter CoreRange. The acceptance of a replicate is not mandatory. L servers may refuse to accept a replicate if they already are overloaded. Hence, the operation may fail. If an L server accepts the replicate, it disseminates the replicate in parallel with its local hints, vertically and possibly horizontally.

The cloning of the exact information on other L servers reduces the lookup load on the original L server. Several vertical paths can be created if CoreRange is used on several contiguous hierarchical layers. As fig. 3 shows, the lookup load is distributed between three vertical paths when CoreRange is active on the first (L^1_h) and second (L^2_b) hierarchical layers. By combining SpreadRange and CoreRange dissemination, L servers are able to control the number of paths available to resolve an *id* autonomously, creating an effective mechanism to handle peaks of load.

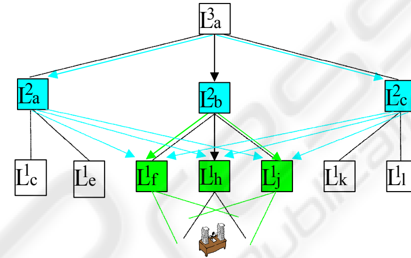

Figure 3: CoreRange parameter

However, core meshes also increment the update overhead. In the special case of the lowest hierarchical layer, the information is the complete information and not a hint. A CoreRange modification may produce a vertical and horizontal dissemination (if SpreadRange is not null) of hints. Therefore, CoreRange will be increased or decreased in result of increases and decreases of the lookup load from far away or upper layer L servers.

When several application server replicas are available in the neighborhood, L server must distribute the load uniformly amongst them. Once more, this can only be achieved if all L servers with those hints use the same CoreRange value. L servers in the SpreadRange region forward lookups to L servers in the CoreRange region. The former servers have to know most of the latter to be able to distribute the lookups evenly. Therefore, the SpreadRanges of the L servers in the CoreRange region must cover more or less the same area. It can be proved that load can be balanced for a network with a constant number of neighbors, if L servers within SpreadRange select the forwarding L server within the CoreRange using a weight distribution algorithm, where weights decrease geometrically with the distance in a factor inversely proportional to the average number of neighbor L servers. On an unknown network it is better to use a lower factor – it is better to concentrate the load in the border of the CoreRange region than in the interior, because probably there will be more L servers at the border. With these mechanisms the original L server

receives less lookup requests from the SpreadRegion per L server, but receives from all directions.

C. L server segmentation and hierarchy

The final mechanism to handle load peaks is the creation of extra L servers at the same hierarchical layer. It reduces the number of application servers registered on some of the L servers. Additionally, if the creator L server is within the CoreRange region of another server, it increases the total processing capacity of that region, until the maximum compute power available.

If the density of L servers on a hierarchical layer becomes too high (compared to the border layers), the L server clustering algorithm may react creating new hierarchical levels. In result, the number of hierarchal levels is not uniform throughout the network: crowded areas can have deeper branches than other less crowded areas.

D. Load adaptation algorithm

An inter-L server co-ordination algorithm is used to guarantee that L servers are enough to respond to the requests and to keep the hints coherent during internal modifications. It is assumed that a percentage of the bandwidth is always available for control and signaling functions. The algorithm reacts to load measurements and registrations of identifiers using four main parameters: SpreadRange; CoreRange; the number of L servers at each hierarchical layer; and the number of hierarchical layers.

L servers monitor their local lookup load, determining if the lookups were routed from lower layer or "near" neighbors L servers (*FromDown*), or if they were routed from higher layer or "distant" L servers (*FromUp*). L servers measure their average load on fixed length intervals, using (1) (a modified discrete first order filter to attenuate the variation of the load measured). Coefficient α_i has two different values whether the load increased or decreased (α_{up} and α_{down}) compared to the previous average value. In result, the algorithm reacts more promptly to raises of the load. The load algorithm makes the system react in situations of overload (a threshold value of *MaxLoad*) and underload (*MinLoad*). L servers also monitor their queue, and react when it increases above a threshold value (*1stMaxQ*). This second mechanism speeds up response for sudden raises of load. Each time there is a reaction, a minimum interval time (*MinPeriod*) is defined to prevent a second reaction. When this time expires, the adaptation is fired again if the queue length increases above a new threshold. The new threshold takes into account the clients in the queue plus a constant increment (*MQinc*). The increase is done in such a way that the more loaded the system is (the delay increases and so does the queue) the greater is the sensitivity of the threshold (in relative terms the value has decreased) making the whole system react more often.

$$load_n = \alpha_i . measurement_n + (1-\alpha_i) . load_{n-1} \quad (1)$$

When an overload trigger happens in an L server, it tries to:

```
1. if FromDown then
      Increase SpreadRange on the lower layer
      L servers;
   if FromUp then
      Increase local CoreRange;
2. If 1 failed and (load_n > NewReplicaLoad)
      Segment L server; if violates density,
      modify hierarchy
```

When an underload trigger happens in an L server, it first tries to reduce SpreadRange and CoreRange, turning the system into a more pure hierarchy. Afterwards, if $load_n$ goes below a minimum threshold, the L server tries to self-destruct. Before, it runs an agreement protocol to assess that all neighbor L servers are unloaded and select one of them to receive its lower layer L servers.

The location service parameters are also influenced by application server updates. If a server changes its location frequently the SpreadRange and CoreRange parameters will be reset frequently to zero, and in consequence, the hint dissemination is almost restricted to the vertical dimension, with low update overhead. A bigger number of application replicas produce a more uniform *id* hint distribution through the L server network, concentrating load on the lower hierarchical levels. This characteristic allows a good adaptation to extreme load peaks.

Compared to the approaches analyzed is section 3, the proposed location service presents search and update costs comparable to DNS for low load levels. When load increases, update costs are increased by a value proportional to the number of hierarchical levels (*h*), the extra number of paths created, and the optional horizontal load balancing costs. Notice, however, that update costs are reduced by the use of chained hints, which restrict the high priority update area to the first hierarchical L servers with complete reference information (except for deletions, which will seldom occur during overload).

6 SIMULATIONS

The location service presented on this paper was simulated using the Ptolemy simulation system (Ptolemy). The simulator implements a dynamic

application, where servers measure their request queue and react to local overload creating application server clones, in a number proportional to the ratio between the growth rate of the queue and the average service time. We assumed that application servers run in parallel without inter-synchronization. The complete application algorithm is described in (Bernardo, 1998).

Clients make a resolve request of the application *id* on the nearest L server and treat the response. The response is either a definitive one and the application server is invoked, or is a reference to another L server in which case the client repeats the process. If an L server takes more than 0.5 tics of simulation time to respond, clients go back to the previous L server. It is assumed that delays are due to load, and new application servers could appear in the meanwhile allowing the client to get a fresh one.

Application servers run one client at a time during the service time (S), and keep the remaining requests on a queue. If the application server takes more than 1.5 tics to answer to a client, the client goes back to the location service and tries to locate another application server. Again, this procedure allows clients to deal with location service adaptation, using new fresh paths that could appear.

All simulations were conducted with a network of 625 compute nodes where application and L servers run. Each node runs a LP that has an average of three connections to its neighbors, and the network has a maximum distance of 24 node hops. At time zero the location service has three hierarchical layers, with 75 L servers at the first layer, five L servers at the second layer and a single root L server at the third layer. Simulations evaluate the behavior of the location service when, at instant one (tic), a total load of 625 application clients per tic (uniformly distributed on the network) try to run the application with a single starting application server. During the simulation, L servers adapt to the load. They measure the processor utilization time during intervals of 0.5 tics and test the average load after each measurement interval (using 75% for α_{up} weight and 50% for α_{down}). *MaxLoad* and *MinLoad* thresholds were respectively 95% and 0.2%. L servers also monitor the lookup queue lengths, reacting with parameters *MinPeriod* and *MQinc*, respectively 0.2 tics and ten clients. *1stMaxQ* depended to the L server lookup service time.

In order to test scalability, the lookup service time ($1/\mu_L$) took six values raging from 1 mtic (1000 lookups per tic) to 20 mtics (50 lookups per tic). *1stMaxQ* was set to μ_L. Longer values put more stress on the location service. The largest value requires that at least thirteen L servers have references to the application server, to handle the load. Experiments with the minimum value showed that a pure hierarchy could do the job.

Three kinds of application behavior were tested: 1 - a static application server with a service time of 0.001 tics; 2 – an adaptive application behavior with service time of 0.01 tics, which originated a small set of application servers (~10); 3 - an adaptive application behavior with a service time of 0.1 tics, which originated a large set of application servers (~100). For situations 2 and 3, the time needed to create a clone of an application server was set to one tic. The scalability of the proposed algorithm can be proved by the time it takes to reduce the number of client lookups waiting on the L servers queues and the total number of clients waiting on the system (both L server queues and application server queues), respectively $tStab_L$ and *tStab*, to values lower than the rate of incoming clients (312 elements). Figure 4 shows that for every combination tested the location service stabilized. $tStab_L$ does not have a strong relation with the value of $1/\mu_L$, except for the highest values tested with a single application server. The faster reactions of the queue length triggered mechanism for loaded systems compensated the extra adaptations, showed in figure 5. On behavior 1 (single application server) a pure tree would become overloaded for values of $1/\mu_L$ above 1.6 mtics. Above this value, L servers at second and third layer became overloaded, and set SpreadRange on the first layer to its maximum allowed value (6). The value of CoreRange increased with the growth of $1/\mu_L$, reaching its maximum allowed value for $1/\mu_L$ = 20 mtics (where two additional L servers were created on the first hierarchical layer). For behaviors 2 and 3 the creation of extra application servers helped to disseminate the *id* lookup load amongst several branches of the location service hierarchy, reducing the final values of CoreRange and SpreadRange.

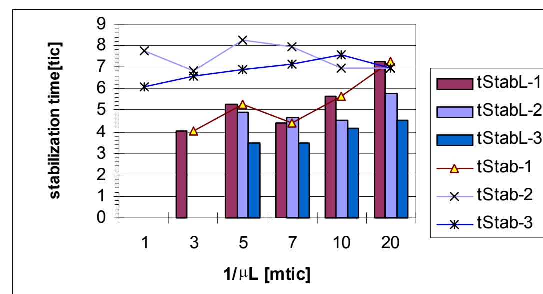

Figure 4: Stabilization times for the location service ($tStab_L$) and the total system (*tStab*)

Notice that *tStab* is almost independent of both the lookup and the application service times when

dynamic application server deployment is used. When the location service adaptation time is slower, the location service accumulates more clients on the L server queues. Once L servers react these clients are received on the application servers at a larger rate, creating a larger number of application server clones. In consequence, these clients will be processed at a higher rate.

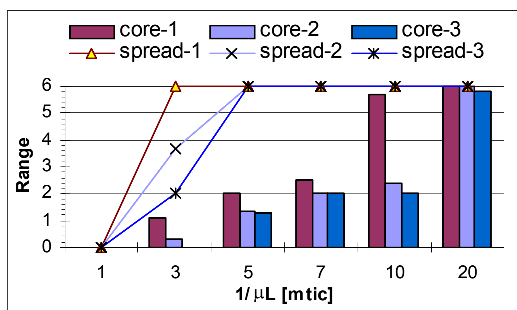

Figure 5: CoreRange and SpreadRange values at instant 100 for first layer L servers with application registrations

7 CONCLUSIONS

Location services will play a very important role on future Internet services and applications. This paper evaluates some of the most important contributions from the P2P and grid community to solve the problem, and proposes solutions to two unhandled problems: the tree root bottleneck and the dynamic control of information replication. Previous services relied on a fixed number of paths and nodes to handle search load. Caching solved much of the overload problems however it also prevented the applications adaptation during the lifetime of cached values. The proposed service adapts to the load, creating and destroying paths on demand, in order to have the minimum update and search overhead. The update costs are reduced by the use of chained hints and by deploying just enough search paths to handle the load. Simulations results show that the solution scales with the relative increment of the load and is capable of handling concurrent search and update load.

Several other aspects of scalability could not have been addressed here: how does the entire system cope with the increase on the number of the identifiers? What are the consequences of slow and fast mobility for the coherence of the information on the servers? How large can the exchange of data be when the system is loaded due to several identifiers making the SpreadRange and the CoreRange raise toward their maximums? Another investigation subject is the support for ad hoc wireless networks. This proposal assumes that the core LP network changes seldom. For ad hoc we are investigating a new overlay structure that improves the performance of search based approaches.

REFERENCES

Adamic, A. L., Huberman, B., 2002. Zipf´s law and the Internet. In *Glottometrics* Nº 3, In *http://www.ram-verlag.de*.

Beck, M., Moore, T., 1998. The Internet2 Distributed Storage Infrastructure Project: An Architecture for Internet Content Channels. Computer Networks and ISDN systems, Vol. 30, pp. 2141-2148, Nov.

Bernardo, L., Pinto, P., 1998. Scalable Service Deployment using Mobile Agents. In MA'98, the 2nd International Workshop on Mobile Agents, LNCS Vol. 1477, Springer Press.

Bernardo, L., Pinto, P., 1998b. A Scalable Location Service with Fast Update Responses. In Globecom'98, IEEE Press.

Chawathe, Y., et al., 2003. Making Gnutella-like P2P Systems Scalable. In *SIGCOMM'03, the 2003 conference on Applications, technologies, architectures, and protocols for computer communications* ACM Press.

Czajkowski, K., Fitzgerald, S., Foster, I., Kesselman, C., 2001. Grid Information Services for Distributed Resource Sharing. In HPDC'10, *the 10th IEEE International Symposium on High-Performance Distributed Computing*, IEEE Press.

Foster, I., et al.., 2002. Grid Services for Distributed System Integration. IEEE Computer Vol.35, pp.37-46, June.

Gummadi, K., et al, 2003. The Impact of DHT Routing Geometry on Resilience and Proximity. In *SIGCOMM'03,* ACM Press.

Gupta, R., Talwar, S., Agrawal, D., 2002. Jini Home Networking: A Step toward Pervasive Computing. *IEEE Computer*, Vol. 36, pp. 34-40, Aug.

Hildrun, K., Kubiatowicz, J. D., Rao, S., Zhao, B. Y., 2002. Distributed Object Location in a Dynamic Network. In SPAA*'02, 14th annual ACM symposium on Parallel algorithms and architectures*, ACM Press.

Partridge, C., Mendez, T., Miliken, W., 1993. Host Anycasting Service. IETF RFC 1546.

Perkins, C., Belding-Royer, E., Das, S., 2003. Ad hoc On-Demand Distance Vector (AODV) Routing. IETF RFC 3561.

Ptolemy home page, http://ptolemy.eecs.berkeley.edu

Rowstron, A. I. T., Druschel, P., 2001. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In Middleware'01*, 18th*

IFIP/ACM Int. Conf. on Distributed Systems Platforms, LNCS Vol. 2218, Springer Press.

Schollmeier, R., Schollmeier, G., 2002. Why Peer-to-Peer (P2P) does scale: An analysis of P2P traffic patterns. In P2P'02, 2nd Int. Conf. on P2P Computing, IEEE Press.

Steen, M., et al.., 1998. Locating Objects in Wide-Area Systems. *IEEE Communications*, Vol. 36, pp. 104-109, Jan.

Vakali, A., Pallis, G., 2003. Content Delivery Network: Status and Trends. *IEEE Internet Computing*, Vol. 7, pp. 68-74, Nov-Dec.

Zhao, B. Y., et al., 2002. Brogade: Landmark Routing on Overlay Networks. In IPTPS'02*, the 1st Int. Workshop on Peer-to-Peer Systems*, Springer Press.