A Comparison of Alternative Web Service Allocation and Scheduling Policies

Esam Alwagait and Shahram Ghandeharizadeh
Department of Computer Science
University of Southern California
Los Angeles, CA 90089, USA
alwagait@usc.edu, shahram@usc.edu

Abstract

Web Services (WSs) are emerging as the building block of Internet scale database management systems (IDBMSs). These systems must intelligently execute plans that reference autonomous WSs. This requires policies and mechanisms for both scheduling and allocating WSs that constitute a plan. In this study, we analyze two scheduling strategies and four allocation policies. Obtained results show that dynamic scheduling with Least Response Time (LRT) allocation policy is superior to other alternatives when the service time of a WS can be estimated accurately. The traditional Least Recently Used (LRU) allocation policy is inferior to all policies including Random. These observations are important because they impact the scalability of a system. Only with a smart allocation policy, one should expect improved system performance by increasing the number of nodes that constitute an IDBMS to support a larger number of WS replicas.

1. Introduction

Web services are the building blocks of Internet scale frameworks for sharing and exchange of data in support of both computation and data intensive tasks. Many envision extensible frameworks that consume a user request to compose a plan that incorporates available web services, and execute the plan seamlessly. WS specifications from both industry, e.g., Advanced Web Services by Microsoft, IBM and others, DBPEL4WS[10], etc., and academia, e.g., XL [9], serve as a foundation to develop these frameworks. One such framework is Proteus [7], a system with a SQL front-end implemented using the Advanced Web Services (see http://msdn.microsoft.com/webservices/understanding/advancedwebservices/default.aspx) specification. These frameworks have the potential to realize Internet scale extensible DataBase Management Systems (IDBMSs) where a web service is a plug-in module. It shares similar challenges with a traditional Extendible DataBase Management Systems (EDBMSs), e.g., Postgress [14], such as how to compose efficient plans, mechanism to index data for efficient processing, secure access to data and services, high level user interfaces for data retrieval and processing, etc. A key difference between an IDBMS and an EDBMS is that WSs are autonomous modules that are dynamically inserted and removed from a distributed framework. This difference has two ramifications. First, an IDBMS should discover new WSs when they are introduced on the Internet and evaluate their equivalence to either existing or previously available WSs [6,16,4]. This enables an IDBMS to compose plans that references relevant web services. Second, an IDBMS should provide for fault tolerant processing of plans and utilize smart distributed techniques when executing a query. This enables an IDBMS to provide high performance.

Proteus as an IDBMS has many diverse applications. Here, we provide two examples. With the first, imagine an application of Proteus that produces a map of businesses in a certain geographical area corresponding to a business category such as “Automobile Tires”. Proteus might utilize Yahoo yellow pages (YP), GeoCoder(GC), and TigerLines (TL) WSs to compose a plan. It glues these web services together using two classes of internal WSs that provide relational algebra operators, see Section 3. A second example comes from the field of neuroscience which is observing a tremendous Internet growth with sites such as PubMed publishing their WSs. As a small example, scientists investigating Parkinsons focus on a region of brain termed Caudoputamen. A neuroscientist might pose the following query: What are the axonal connections of the Caudoputamen, and what are definitions of the Caudoputamen and closely related terms? [15] There are several candidate data sources for processing this query. BAMS (http://brancusi.usc.edu/bkms) and the UMLS Meta thesaurus (http://www.nlm.nih.gov/pubs/factsheets/umls.html) are two relevant WSs. Once again, Proteus glues these WSs into a plan using its own internal WSs.

A plan might be either a linear sequence or a forest of WSs. A forest consists of a bushy collection of linear sequences realized using the Branch, Split, Join, and Union WSs. An IDBMS might execute a plan in either a centralized or a decentralized manner. Both require an IDBMS to (1) allocate a copy of each WS that constitutes the plan, and (2) schedule the plan for execution. The demand for a WS, e.g., Google, might be so high that it is replicated on different servers. In this case, the first item chooses between the different copies of a WS. Ideally, the fastest copy of a WS is allocated to a plan and it is scheduled in a manner that minimizes response time.

A plan might be scheduled either statically or dynamically. With a static allocation, the WSs that constitute a plan are allocated when the plan is submitted for execution. This makes plan execution simpler, since intermediate WSs do not participate in node allocation. However, this reduces the efficiency of operator allocation criteria because the information used to make the decision might be outdated after the execution of the plan starts. With a dynamic allocation, the WSs are allocated on demand by conceptualizing a producer/consumer relationship between WSs. A consuming WS is allocated before a producing WS finishes execution. This requires intermediate nodes to participate in selecting the copy of next WS. However, the decisions are more accurate and up-to-date, since the information in the lookup directory might have been updated after the plan execution started.

The primary contributions of this study are: four WS allocation policies namely, Least Recently Used (LRU), Least Recently Assigned (LRA), Least Utilized (LU), Least Response Time (LRT), two scheduling policies (static and dynamic), and a comparison of their combinations with one another. As a yard stick, we analyze a random allocation policy. As suggested by its name, this policy allocates a WS copy randomly. Obtained results demonstrate that LRU is inferior to other policies. LRT with dynamic scheduling is superior to all other policies. The other policies perform the same as Random with static scheduling. With dynamic scheduling, their performance improvement relative to Random is marginal. These results are a contribution because they impact scalability of an IDBMS such as Proteus. Section 3 shows that increasing number of node and WS replicas by a factor of four does not improve performance with an inferior allocation policy.

The rest of this paper is organized as follows. Section 2 provides an overview of our target environment and the alternative allocation policies. Section 3 employs a simulation study to investigate alternative combinations of allocation and scheduling policies. Brief conclusions and future research directions are offered in Section 4.

2. A Multi-dimensional framework

From September to December 2003, Proteus was implemented using Microsoft's Web Services Architecture (WSA, formerly known as Global XML Architecture or GXA) [11, 12]. Proteus is a framework that executes plans by utilizing geographically-distributed WSs. The plan is executed one step at a time and the intermediate results are passed from one WS to the next WS until the plan execution is complete. Proteus utilizes WS-Routing and WS-Addressing standards (http://msdn.microsoft.com/webservices/understanding/specs/default.aspx) to implement decentralized query processing. It uses XML to represent a plan and its intermediate results. This representation is encapsulated in SOAP envelopes that are passed between nodes using TCP protocol.

Proteus consists of three main components, namely, the mediator, the lookup registry and the intermediate nodes. The mediator receives a query specified in datalog query language [17] and uses the available data sources to create a plan equivalent to the query. The plan defines the identity of participating data sources, i.e. WSs, required to execute the query along with the input and output schema for each data source. It is, then, inserted in the SOAP header and passed to the first node in the plan. The plan is accessible to all intermediate nodes involved in the plan execution. This enables a WS to identify the next WS to consume and process its output.

The intermediate nodes host WSs which participate in the plan execution. Their role is to use the input data to execute, modify the SOAP header to store the results and then forward the modified SOAP envelope to the next intermediate node. This process is repeated until plan execution is complete. The lookup registry contains information about the available data sources, their input/output schema and their locations. With static scheduling, the mediator queries the registry to build the WS-Routing header with the locations of WSs used in the plan. When using dynamic scheduling, it is the intermediate nodes' responsibility to query the lookup registry and modify the WS-Routing header with the location of the next WS.

The current Proteus implementation is deployed in a controlled lab environment and has several restrictions. First, it assumes the presence of one replica of each WS. Second, it assumes a centralized lookup registry. This motivated the need for policies and mechanisms to allocate and schedule multiple replicas of WSs when available. The centralized lookup registry is not realistic for large-scale deployments of such frameworks. It can be eliminated using the DeW framework [2]. DeW employs CAN [13], a P2P network which is failure-tolerant. The remainder of this section describes how we eliminate the first restriction using DeW.

When multiple copies of a WS are available, we envision using an allocation policy to select a copy. We consider the following four policies: Least Recently Used (LRU), Least Recently Allocated (LRA), Least Response Time (LRT), and Least Utilization (LU). The allocation process in all these policies is accomplished by maintaining information about usage of WSs and the nodes hosting them. This information is termed Performance Metadata (PM). DeW registry stores and retrieves the WS name and its PM as a (key, value) pair. DeW framework can be used as a normal P2P with lookup, i.e. read, and write functionality with WS name as the key. It publishes its own WS that implements a given policy. We term this WS, Allocation-WS. Given the distributed nature of DeW, it can support many replicas of Allocation-WSs. Different WSs that register themselves with DeW can be dispersed across the available Allocation-WSs using a distributed hash table such as LH*, Chord, CAN, etc. Implementation of this important detail imposes additional network overheads. We ignore these for now to observe the best a policy can perform in the absence of DeW overheads. Results presented in Section 3 show that a complete analysis of this overhead with some policies is almost certainly useless because they are simply not competitive, i.e., a simple random allocation policy outperforms them. Thus, we keep the descriptions of these policies brief.

The four allocation policies can be categorized based on whether they allocate WSs based on their individual statistics (termed WS-Stats) or the nodes containing WSs (termed Node-Stats), see Table 1. LRU and LRA are WS-Stats policies. They perform poorly when multiple different WSs are assigned to the same node. LRU considers node statistics. LRT considers WS-Stats and performs better because it considers queuing delays incurred by each WS. Below, we detail these techniques in turn.

Allocation policy	WS Statistics	Node Statistics
LRU	YES	NO
LRA	YES	NO
LU	NO	YES
LRT	YES	NO

Table 1: Four allocation policies

2.1 Least Recently Used (LRU)

This policy maintains a time stamp for each copy of a WS. This policy allocates the copy of a WS which has been least recently used. The time stamp is updated every time a WS finishes executing its currently assigned plan. The intuition is to distribute the load between copies of a WS evenly. A limitation of this policy is its ignorance of the assignment of WS replicas to nodes. For example, replicas of two different WSs assigned to the same node might be allocated simultaneously because both were least recently utilized. An obvious improvement is to extend LRU to consider the utilization of participating nodes, see Sections 2.3 and 4.

2.2 Least Recently Allocated (LRA)

This is similar to the LRU policy, except that the time stamp is updated when the operator is allocated, i.e. before starting the execution. Similar to LRU, LRA does not consider the utilization or service time of the node and suffers from both inter and intra-WS collisions. It can also be extended to consider utilization of nodes, see Sections 2.3 and 4.

2.3 Least utilized (LU)

This policy allocates WSs based on the utilization of their node. It chooses the WS hosted on the least utilized node. Different plans have different input sizes impacting the utilization of each node, which is considered by this policy. The PM for this policy includes the utilization of nodes where copies of a WS are hosted. This policy requires interaction between the node and the DeW registry. It is the responsibility of each node to update its utilization. We envision two ways to do this, namely, updating the utilization either (1) periodically or (2) updating it when processing of a request has commenced and completed (these two events specify node utilization). Deploying DeW as a WS makes it easier to update the utilization.

One may combine LU with either LRU or LRA. These policies are in synergy because LU considers node usage while the other two consider WS usage. Section 4 describes several different ways for combining these techniques. An evaluation of these possibilities is a future research topic.

2.4 Least Response Time (LRT)

This policy estimates the expected service time for each WS and chooses the WS with the Least Response Time. We define response time as the elapsed time from allocating the WS until the WS execution finishes. It is impacted by the hardware speed of the node hosting the WS (service time), and the number of requests waiting in the queue of a referenced WS (queuing delays). This policy updates the PM the same way as LU.

3. Evaluation

We compared the four policies assuming environments consisting of ten, twenty and fourty nodes. Our observations apply to all three configurations. The simulation model ignores the overhead of maintaining the PM information for LRU and LU up to date. This is because these techniques are inferior to Random in the absence of this overhead. We wanted to establish this without the complexity to maintain PM up to date. The performance of these techniques would almost certainly be worse once this overhead is incorporated, i.e. obtained results represent the best performance attainable by LRU and LU.

We have 3 types of queries each represented as a plan. All three queries use the same data sources, namely: Yahoo yellow pages (YP), GeoCoder (GC) and Tigerlines (TL) WSs. The first query extends information provided by YP with the longitude and latitude information of the business address, obtained from GC. The client uses this information to show the location of the business on an aerial map obtained from Microsoft TerraServer [5]. The second query employs TL to add the street information (start and end longitude and latitude) to the result. The third query is the same as the second with one difference: there exist multiple TLs for different states. This means the plan must query each TL individually and union their output.

Each query plan employs Proteus specific WSs that serve as the glue between the autonomous WSs. These are categorized into two groups. The first corresponds to standard relational algebra operators: select, project, join, etc. The second implements data-flow-and-control operators such as Branch and Split. These resemble simpler versions of the Exchange operator [19] and Edies [3, 18]. The Branch WS constructs k copies of its XML formatted input data and forwards a copy to a different pre-specified WS. The Split WS is provided with a selection clause and a destination WS for the qualifying XML elements. It consumes its XML formatted input, applies the selection clause, and forwards all the qualifying elements to the specified target WS. Figure 3-1 shows three different query plans. The first employs a geo-coder to show the location of the business on the map. It employs the Branch operator in combination with Project because the GC WS accepts only the street address, city, state and zip of the business, i.e. phone number and other information cannot pass through. The second plan repeats the same with TL because this WS accepts only the same address information as GC. The third query employs the split operator to send each address to the TL WS in the corresponding state. The results of all states are combined together using the Union WS which performs duplicate elimination.

We used Proteus implementation to execute the three queries in a small lab environment where only one copy of a WS is present. The service times observed from these experiments are plugged into the simulator and used to compare the different combinations of allocation and scheduling policies.

We considered both homogeneous and heterogeneous environments. The homogeneous environment consists of machines with 2.0 GHz processors. The heterogeneous environment consists of a uniform mix of 5 different processor speeds: 1.4, 1.6, 2.0, 2.4, 3.0 GHz. We Assume a fully connected network topology with fixed delay between any two nodes. Figures 3.2-3.5 show the average execution time for Query 3 using static and dynamic allocation techniques with the two alternative environments. The observations from plans 1 and 2 are identical and eliminated from further discussion.

There are two forms of collisions in our environment. The first, termed intra-WS collisions, occurs when the same copy of a WS (say GeoCoder) is utilized simultanteously, resulting in formation of queues. The second, termed inter-WS collision, refers to the scenario where requests reference different WSs (e.g., GeoCoder and Yahoo WSs) hosted on the same node. A collision is further categorized as either inter-plan or intra-plan. With inter-plan collisions, different plans compete for the same node. With intra-plan collisions, branches of the same plan might compete for the same node at the same time. All four possible collision types are captured by our simulator.

We employ an open simulation model where requests arrive at a pre-specified rate (λ) using a Poisson distribution. Since Proteus is a distributed system, a naïve policy that results in may plans colliding on the same node (due to inter and intra-WS/plan collisions) will cause that node to become a bottleneck at a certain arrival rate. This node becomes fully utilized with many queued up requests while other nodes sit idle waiting for work. At this point, the observed execution times are dependent on the implementation of the employed random number generator and are un-reproducible. This simulation state is termed unstable. A technique that supports the highest arrival rate prior to the simulator becoming unstable is superior.

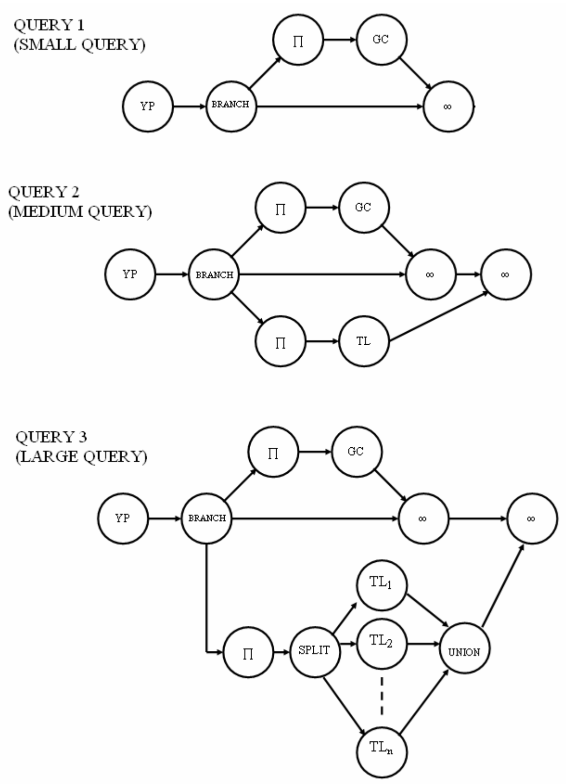

Figure 3-1. Three queries used to evaluate the four policies: Small (Query1), medium (Query2) and large (Query3) queries.

Figures 3.2 to 3.5 show LRU is inferior to all other policies because it becomes unstable at a lower arrival rate. LRT is superior to all other polices when service time is estimated accurately. When compared with dynamic, LRT's performance is inferior with static. This is because static scheduling invokes allocation of all WSs that constitute a plan before the plan execution starts. This means that even though a WS is assigned to the plan at time t_1, it is not actually used till time t_2 ($t_2>t_1$). This increases inter-plan (both inter and intra-WS) collisions, making the incurred service time at time t_2 higher than that estimated at t_1 due to queuing delays.

We were pleasantly surprised to find LRT performing well as long as the estimated response time of a WS is randomly distributed along a mean that matches the true service time of the WS. Figure 3.6 shows LRT's performance with different scheduling policies (static and dynamic) for different random distributions. Given a service time S, a random distribution of 70% corresponds to service times randomly picked from the range S ± 0.7S. Note that LRT's performance is degraded significantly if the estimated response times are completely random.

We also observed that LU performs worse than LRA with static scheduling. This is because LRA does load balancing in a similar manner using the timestamps while LU uses the utilizations measurement which fluctuates frequently due to bursty nature of request arrivals. These fluctuations are because LU assigns many requests to the least utilized node when many requests arrive in a short period of time. This least utilized node remains attractive until its utilization increases once it services its assigned requests [8]. This detective characteristic of LU results in formation of bottlenecks that migrate from one node to another, resulting in significant queuing delays. Compared to LU, LRA does not require any network communication with DeW because it requires no node characteristics to render a decision. This makes LRA more desirable than LU.

We investigated the scalability of these policies and noticed that introducing more nodes in the system does not always improve performance. Figures 3-7 to 3-9 show the performance improvement of LRU, LRT and Random for three different configurations consisting of ten, twenty and forty nodes, respectively. Similar to prior experiments, WSs are replicated across all nodes. LRU performance does not scale because it results in formation of bottlenecks with a low arrival rate. While Random is more scalable than LRU, a configuration deployed with LRT exhibits the best scalability characteristics. Similar trends are observed with both (1) homogeneous configurations and (2) static scheduling strategy.

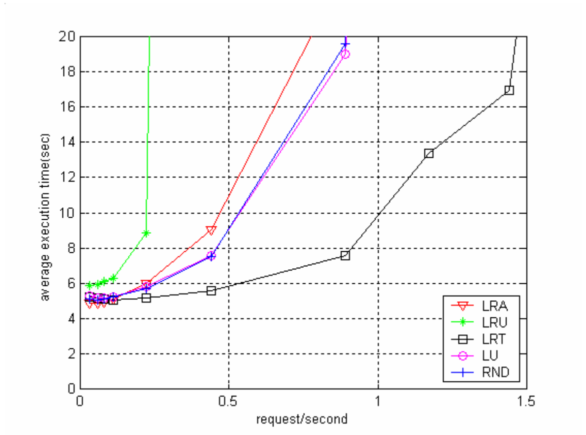

Figure 3-2. Average execution time of Q3 with alternative allocation policies, heterogeneous configuration using dynamic scheduling.

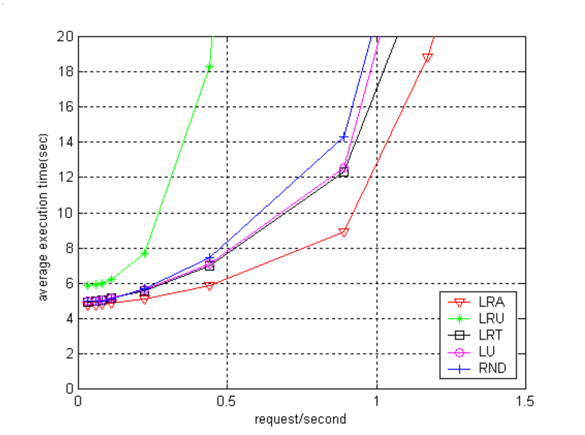

Figure 3-5. Average execution time for Q3 with a homogeneous configuration using static scheduling.

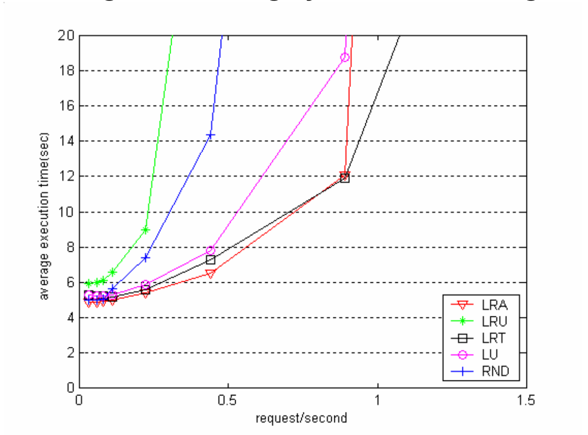

Figure 3-3. Average execution time of Q3 with alternative allocation policies, heterogeneous configuration using static scheduling.

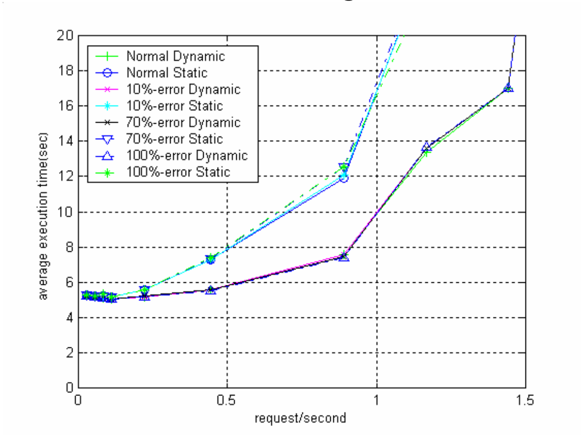

Figure 3-6. Average execution time for Q3 with LRT, heterogeneous configuration using different random distributions.

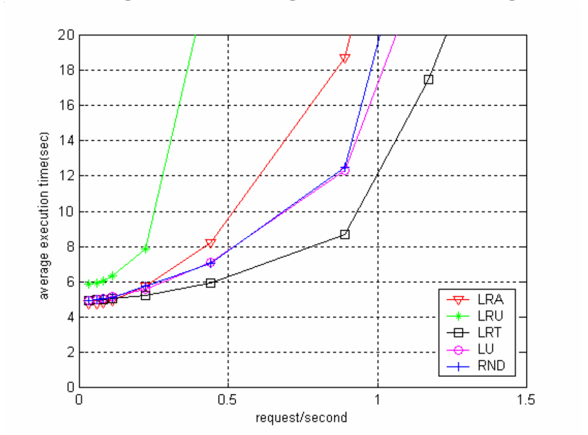

Figure 3-4. Average execution time of Q3 with alternative allocation policies, homogeneous configuration using dynamic scheduling.

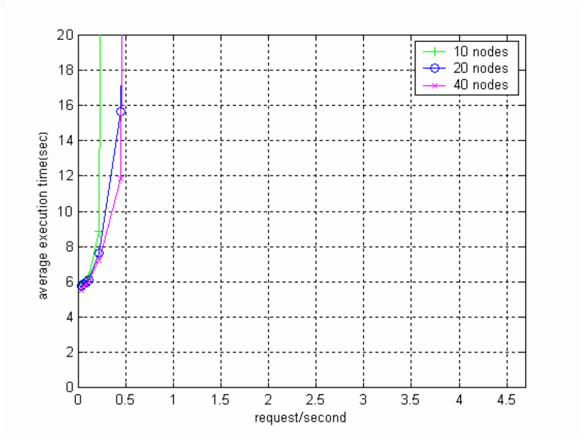

Figure 3-7. Scalability of a heterogeneous configuration with LRU allocation policy and dynamic scheduling.

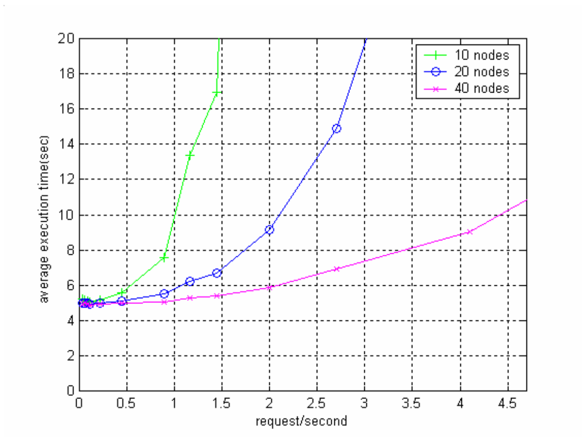

Figure 3-8. Scalability of a heterogeneous configuration with LRT allocation policy and dynamic scheduling.

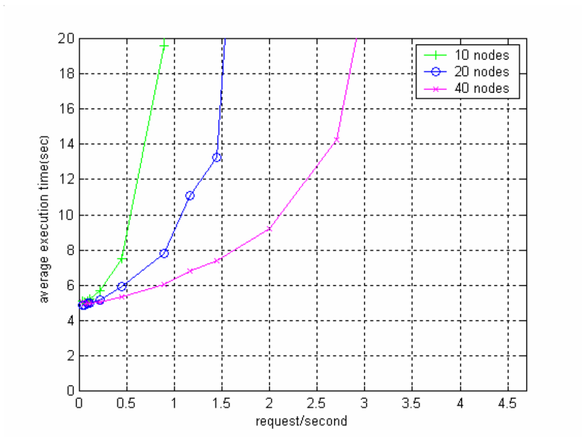

Figure 3-7. Scalability of a heterogeneous configuration with Random allocation policy and dynamic scheduling.

4. Conclusion and Future Research

This paper presents two different scheduling techniques and four alternative allocation policies. Obtained results demonstrate these design decisions have a significant impact on the performance of an IDBMS such as Proteus. The main insights are as follows. First, dynamic scheduling is superior to static. Second, an allocation policy such as LRT is superior to other alternatives. Third, a WS allocation policy must consider the utilization of nodes when multiple copies of different WSs are assigned to the same node. In our experiments, LRU and LRA do not capture this important detail, providing inferior performance when compared to other policies (including a Random allocation policy).

We intend to extend this preliminary study in several ways. First, we are using lessons from our evaluation study to develop better allocation policies. For example, one may extend LU to consider the load imposed by new requests that are assigned to an underutilized node [8]. This might enable LU to support a higher arrival rate. Moreover, LRT might not be realistic for those environments where response time cannot be estimated accurately. In this case, a hybrid of LU and LRU might be more realistic. In general, an ideal allocation policy must consider both node and WS profiles. Second, we intend to model the overhead of updating the PM of a WS once registered with DeW. This requires extensions to our simulation model. Finally, we are considering techniques to create and destroy WS replicas on-demand. Ideally, these techniques must construct (destroy) additional copies of a WS when their utilization is high (low). This is particularly important for Proteus specific WSs such as join, project, branch, union and split. One possible implementation is to extend DeW in support of this functionality.

5. Acknowledgments

This research was supported in part by an unrestricted cash gift from Microsoft Research, NSF grant IIS-0307908, and National Library of Medicine LM07061-01.

6. References

[1] B. Albahari, P. Drayton, and B. Merrill. C# Essentials. O'Reilly, 2001.

[2] E. Alwagait and S. Ghandeharizadeh. DeW: A Dependable Web Services Framework.. In 14th International Workshop on Research Issues on Data Engineering, Boston, MA, March 2004.

[3] R. Avnur and J. Hellerstein. Eddies: Continuously Adaptive Query Processing. In Proceedings of ACM SIGMOD 2000.

[4] F. Banaei-Kashani, C. Chen, and C. Shahabi. WSPDS: Web Services Peer-to-Peer Discovery Serivce. In the International Symposium on Web Services and Applications (ISWS'04), June 2004.

[5] T. Barclay, J. Gray, E. Strand, S. Ekblad, J. Richter, "TerraService.NET: An Introduction to Web Services," MSR TR 2002-53, pp 13, June 2002.

[6] S. Decker, S. Melnik, F. Harmelen, D. Fensel, M. Klein, J. Broekstra, M. Erdmann and I. Horrocks. The Semantic Web: The Roles of XML and RDF. IEEE Internet computing, 15(3), 2000.

[7] S. Ghandeharizadeh, C. Knoblock, C. Papadopoulos, C. Shahabi, E. Alwagait, J. Ambite, M. Cai, C. Chen, P. Pol, R. Schmidt, S. Song, S. Thakkar, and R. Zhou. Proteus: A System for Dynamically Composing and Intelligently Executing Web Services. In the First International Conference on Web Services(ICWS), Las Vegas, Nevada, June 2003.

[8] M. Keidel, S. Seltzsam, and A. Kemper. Reliable Web Service Execution and Deployment in Dynamic Environments. In 4th International Workshop on Technologies for E-Services (TES), Berlin, Germany, 2003.
[9] D. Kossmann D. Florescu, A. Grnhagen. XL: A Platform for Web Services . In Conference on Innovative Data Systems Research (CIDR), January 2003.
[10] J. Klein F. Leymann S. Thatte F. Curbera, Y. Goland and S.Weerawarana. Business Process Execution Language for Web Services, Version 1.0. July 2002.
[11] Microsoft Corporation. Microsoft .NET Framework.
[12] Microsoft Corporation. Global XML Web Services Architecture (GXA), 2002.
[13] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content Addressable Network. In ACM SIGCOMM, 2001.
[14] M. Stonebraker. The design of the Postgress storage system. In Proceedings of the 13th VLDB Conference, 1987. Brighton, UK.
[15] L. Swanson. Personal Communications. January 2004.
[16] K. Sivashanmugam, K. Verma, A. Sheth, and J. Miller. Adding Semantics to Web Services Standards. In Proceedings of the International Conference on Web Services, 2003.
[17] S. Thakkar and C. A. Knoblock. Efficient Execution of Recursive Integration Plans. In Proceedings of 2003 IJCAI Workshop on Information Integration on the Web, August 2003.
[18] F. Tian and D. DeWitt. Tuple Routing Strategies for Distributed Eddies. In Proceedings of VLDB 2003, Berlin, Germany, September 2003.
[19] G. Graefe. Encapsulation of Parallelism in the Volcano Query Processing System. In Proceedings of ACM SIGMOD 1990.